UNIVERSIDADE FEDERAL DO PARANÁ

LUIS HENRIQUE ALVES LOURENÇO

PARALELIZAÇÃO DO DETECTOR DE BORDAS CANNY PARA A BIBLIOTECA ITK UTILIZANDO CUDA

CURITIBA

2011

LUIS HENRIQUE ALVES LOURENÇO

PARALELIZAÇÃO DO DETECTOR DE BORDAS CANNY PARA A BIBLIOTECA ITK UTILIZANDO CUDA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Daniel Weingaertner

CURITIBA

2011

L892 Lourenço, Luis Henrique Alves
Paralelização do detector de bordas Canny para a Biblioteca ITK utilizando Cuda / Luis Henrique Alves Lourenço. – Curitiba, 2011.
70f. : il., tabs.

Impresso.
Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Daniel Weingaertner.

1. Computação gráfica. 2. Processamento de imagens. I. Weingaertner, Daniel. II. Título.

CDD: 006.6

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Luis Henrique Alves Lourenço, avaliamos o trabalho intitulado, *"PARALELIZAÇÃO DO DETETOR DE BORDAS CANNY PARA A BIBLIOTECA ITK UTILIZANDO CUDA"*, cuja defesa foi realizada no dia 07 de abril de 2011, às 13:00 horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 07 de abril de 2011.

Prof. Dr. Daniel Weingaertner
DINF/UFPR – Orientador

Prof. Dr. Bruno Schulze
LNCC – Membro Externo

Prof. Dr. Eduardo Todt
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

DEDICATÓRIA

Dedico este trabalho,

À Sarah, pelo apoio em todos os momentos.

Aos meus pais José e Celeste que muito fizeram por mim.

À minha irmã Ana por toda a amizade desde sempre.

AGRADECIMENTOS

Ao Prof. Daniel que acreditou em mim e muito me ensinou.

Aos demais professores que, direta ou indiretamente, contribuíram para a realização deste trabalho.

Aos colegas de laboratório que compartilharam comigo os primeiros passos do grupo de Visão, Robótica e Imagens (VRI).

Aos amigos que fiz em Curitiba. Afinal não foi fácil deixar tudo para trás e começar uma nova jornada.

O presente trabalho foi realizado com o apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq - Brasil.

EPÍGRAFE

”First, solve the problema. Then, write the code.”
(John Johnson)

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
C3SL	Centro de Computação Científica e Software Livre
CITK	Cuda Insight Toolkit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
ECC	Error Check and Correction
GPGPU	General Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
GUI	Graphical User Interface
ILP	Instruction Level Paralelism
ITK	Insight Segmentation and Registration Toolkit
JPG	Joint Photographic Experts Group
MIMD	Multiple Instruction stream Multiple Data stream
MP	Multiprocessadores
MRI	Magnetic Resonace Image
NA-MIC	National Aliance for Medical Image Computing
NCBC	National Center of Biomedic Computing
NMS	Non-Maximum Supression
PNG	Portable Network Graphics
RO	Read-Only
SIMD	Single Instruction stream Multiple Dada stream
SIMT	Single Instruction Multiple Thread
SNR	Sound to Noise Ratio
SP	Stream Processors
VPAC	Victorian Partnership for Advanced Computing
VRI	Visão, Robótica e Imagem
VTK	Visualization Toolkit

LISTA DE SÍMBOLOS

BD	Borda Definitiva
BP	Borda Possível
FN	False Negative
FP	False Positive
NB	Não-Borda
N_{Bl}	Quantidade de Blocos de uma Grid
N_{Px}	Quantidade de Pixels da Imagem
N_{Th}	Quantidade de Threads por Bloco
P_{co}	Porcentagem de Pixels de Borda Detectados Corretamente
P_{fa}	Porcentagem de Falsos Alarmes
P_{nd}	Porcentagem de Pixels de Borda Não Detectados
T_h	Higher Threshold
T_l	Lower Threshold
TP	True Positive

RESUMO

Aplicações de Processamento de Imagens podem exigir poder de processamento tão alto que a computação tradicional não é capaz de fornecer. Uma alternativa eficiente é a computação de Propósito Geral em Placas Gráficas (GPGPU). CUDA é a API da NVidia que implementa o modelo de programação em placas gráficas. Muitas aplicações que foram reimplementadas em CUDA estão alcançando ganhos significativos de desempenho. Este trabalho tem como objetivo aproveitar o processamento paralelo das placas gráficas através do modelo de computação CUDA para proporcionar melhor desempenho ao detector de bordas Canny na biblioteca de processamento de imagens ITK. Para isso, é apresentado um estudo sobre as arquiteturas CUDA e ITK, conceitos pertinentes e as abordagens utilizadas para implementar filtros ITK para executar em placas gráficas. Além do detector de bordas Canny, foram desenvolvidos o cálculo de gradiente Sobel e a Suavização Gaussiana, assim como uma classe de configuração CUDA para o ITK. O desempenho desses filtros foram avaliados mostrando ganhos em qualquer arquitetura de placa gráfica da NVidia. Além disso, técnicas eficientes de programação foram propostas e avaliadas nas arquiteturas de placas da NVidia G80, GT200 e Fermi.
Palavras-chave: Computação em Placas Gráficas; GPGPU; CUDA; Processamento de Imagens; ITK; Canny; Suavização Gaussiana; Convolução; Sobel.

ABSTRACT

Image Processing applications might demand a high processing power that single-core computing is not able to deliver. A cheap alternative is the General Purpose computation on Graphics Processing Units (GPGPU). CUDA is the NVidia API implementation of the Graphics Processing Units programming model. Many applications reimplemented with CUDA are achieving significantly performance gains. This work aims to take advantage of the parallel processing capability of the GPU through CUDA computing model to provide better performance for Canny edge detection filter from ITK processing image library. To do so, we present a study on the CUDA and ITK architectures, pertinent concepts, and the approaches that have been used to implement ITK filters on GPU. Besides the Canny edge detection filter, Sobel gradient computation and Gaussian smoothing filter were implemented, as well as a CUDA configuration class for ITK. The performance of these filters was evaluated, showing a significant speedup on any NVidia GPU architecture. Furthermore, efficient programming techniques were proposed and evaluated on the NVidia Graphics architectures G80, GT200 and Fermi.
Key-words: GPU Programming; GPGPU; Graphics Processing Units; CUDA; Image Processing; Canny Edge Detection Filter; Gaussian Blur; Convolution; Sobel.

1 INTRODUÇÃO

O Processamento de Imagens, em muitos casos, exige uma capacidade intensa de computação, especialmente quando aplicado à área médica. Nesse sentido os algoritmos de processamento de imagens médicas evoluíram nas últimas quatro décadas de forma revolucionária auxiliando diagnósticos em diversas áreas da medicina (BANKMAN, 2009). Porém, ainda hoje muitos algoritmos podem demorar horas para executar, como é o exemplo de implementações tradicionais do algoritmo de classificação de estroma (GURCAN et al., 2007).

Algumas ferramentas se destacam no Processamento de Imagens. Entre elas, o Insight Segmentation and Registration ToolKit (ITK) (INSIGHT..., 1999; IBÁÑEZ et al., 2005) é um *toolkit* de código aberto[1] que possui um conjunto de bibliotecas muito utilizado no Processamento de Imagens Médicas. Mantido pela Kitware Inc., o ITK disponibiliza diversos algoritmos para a criação de fluxos de processamento de análise de imagens, e utiliza os conceitos da programação orientada a objetos para simplificar e flexibilizar a implementação de fluxos.

Apesar do constante crescimento do poder computacional nas últimas décadas, muitas aplicações ainda exigem mais processamento que o disponível pelos processadores tradicionais. Com o alcance da limitação física do paralelismo em nível de instrução (ILP) dos processadores e de sua miniaturização[2] (GHULOUM, 2009; BORKAR, 1999), passou-se a considerar o uso de vários processadores ou núcleos de processamento (*cores*) em uma mesma pastilha para suprir a demanda pela computação de alto desempenho. Neste aspecto a programação em placas gráficas (*Graphics Processing Unit*, ou GPU) tem se destacado com resultados expressivos. Porém, apenas após o desenvolvimento de modelos de programação em placas gráficas voltados para a programação de propósito geral, ou GPGPU (*General Purpose computing on Graphic Processing Units*) (GENERAL-PURPOSE..., 2002), que esse tipo de dispositivo se tornou realmente atrativo para a maioria das aplicações.

Um dos primeiros modelos de programação em placas gráficas a investir na programação

[1] O ITK é licenciado sob a licensa BSD que permite uso irrestrito, inclusive em produtos comerciais.

[2] A miniaturização dos processadores alcançou o ponto onde os problemas de alimentação e dissipação do calor tornaram o avanço nesse sentido proibitivo (BORKAR, 1999).

de propósito geral foi o *Compute Unified Device Architecture* (CUDA) (NVIDIA..., 2010b). Desenvolvido pela NVidia, o modelo CUDA permite ao programador utilizar o paralelismo das placas gráficas para implementar aplicativos de diversas áreas, desde a codificação de vídeo e áudio, até a exploração de gás e petróleo, modelagem de produtos, processamento de imagens na área médica e pesquisa científica (CUDA..., 2007).

O crescimento do volume dos dados pode ser considerado proporcional ao crescimento da capacidade de armazenamento. E assim as imagens podem armazenar cada vez mais informações, como o uso de pixels com profundidades[3] maiores, imagens coloridas[4], resoluções maiores, ou ainda imagens em três ou mais dimensões. A maioria dos processadores de propósito geral possuem capacidade de processamento suficiente para muitos dos algoritmos de processamento de imagens que executam sobre esses dados. Porém, o paralelismo das placas gráficas abriu uma oportunidade para o processamento de imagens ou ainda para a aplicação de filtros de processamento de imagens nos *frames* de um vídeo em tempo real, como por exemplo na segmentação de vídeo (GÓMEZ-LUNA et al., 2009).

1.1 OBJETIVOS

Os objetivos deste trabalho são: (1) Implementar o filtro de detecção de bordas Canny usando o modelo de programação em placas gráficas CUDA integrado à biblioteca de processamento de imagens ITK. (2) Propor técnicas de programação eficientes para o processamento de imagens com ITK e CUDA. (3) Avaliar o desempenho do filtro implementado em comparação com a implementação do Canny encontrada no ITK.

1.2 DIVISÃO DO TRABALHO

No capítulo 2 será apresentada uma revisão bibliográfica sobre o processamento de imagens médicas, o processamento paralelo de imagens, a programação de propósito geral em placas gráficas, implementações de algoritmos de processamento de imagens médicas em CUDA, o algoritmo Canny e implementações dele usando CUDA, e uma discussão sobre a integração entre CUDA e ITK. Seguido pelo capítulo 3 sobre são descritas as implementações realizadas neste trabalho. O capítulo 4 descreve os computadores, as bases de dados e os testes realizados. Os resultados dos testes encontram-se no capítulo 5 que é seguido pela sua discussão

[3] A profundidade de um pixel representa a quantidade de bits utilizados para representar um pixel em uma camada de cor.

[4] As imagens coloridas, em geral, armazenam mais de uma camada de cor onde a mistura das três camadas em um pixel compõe a cor que o pixel representa. Um exemplo são imagens em RGB.

no capítulo 6. O capítulo 7 fecha o trabalho com as conclusões e trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

A integração de filtros implementados em CUDA com um fluxo de processamento de imagens em ITK utiliza alguns conceitos que serão aprofundados neste capítulo, assim como as principais ferramentas envolvidas neste trabalho e os trabalhos relacionados.

2.1 PROCESSAMENTO DE IMAGENS MÉDICAS

Segundo Bankman (2009), a descoberta de fenômenos físicos como os raios-X, ultrassom, radioatividade, ressonância magnética, e o desenvolvimento de instrumentos de imagens permitiram a criação de algumas das ferramentas de diagnóstico mais eficientes da medicina na atualidade. Esses sistemas também são usados para planejar tratamentos e cirurgias, assim como para imagens biológicas. Porém, ainda segundo o mesmo autor, conjuntos de dados em duas, três ou mais dimensões transmitem informações cada vez mais amplas e detalhadas para aplicações clínicas e de pesquisa. Toda essa informação deve ser interpretada a tempo e corretamente para beneficiar a assistência médica. Para auxiliar na interpretação visual de imagens médicas foram desenvolvidas diversas técnicas automatizadas com seus méritos, limitações e domínios de aplicação. Entre elas podemos destacar algumas como o melhoramento; a segmentação; a quantificação; o registro; a visualização; a compressão, armazenamento e comunicação de imagens.

Para intensificar o uso da computação em pesquisas nos Estados Unidos, foi incentivada a criação do programa de Centros Nacionais de Computação Biomédicas (NCBC) (PIEPER et al., 2006). Na criação dos NCBC enfatizou-se a necessidade de soluções de código aberto que pudessem ser modificadas e estendidas para uso acadêmico e comercial. Entre os sete centros criados, a Aliança Nacional para a Computação de Imagens Médicas (*National Aliance for Medical Image Computing*, ou NA-MIC) desenvolve um conjunto de softwares para abordar os diferentes aspectos do processamento de imagens. Esse conjunto consiste em uma plataforma aberta para o processamento de imagens que inclui como ferramentas de programação o *Insight Toolkit* (ITK) para a segmentação e registro de imagens, e o *Visualization Toolkit* (VTK) para

suportar a visualização, a renderização interativa e a manipulação. Um conjunto de componentes GUI customizados para o desenvolvimento de aplicações baseadas no VTK, conhecido por KWWidgets, o 3D-Slicer como aplicação para usuários finais, além de um sistema de infraestrutura (PIEPER et al., 2006).

2.1.1 Insight Segmentation and Registration ToolKit

O ITK (Insight Segmentation and Registration Toolkit) é um conjunto de ferramentas para processamento, segmentação e registro de imagens. Sua criação foi financiada em 1999 pela Biblioteca Nacional de Medicina do Instituto Nacional de Medicina dos Estados Unidos[1] e desenvolvida pelo *Insight Software Consortium*, um consórcio de três parceiros comerciais e três universidades contratados para desenvolver um conjunto de ferramentas de código aberto para o registro e segmentação de imagens multidimensionais. Muito usado no processamento de imagens médicas, o ITK, é um sistema de código aberto, multiplataforma, desenvolvido para ser de fácil aprendizado através de conceitos básicos da programação orientada a objetos (IBÁÑEZ et al., 2005). Os objetivos do ITK incluem (INSIGHT..., 1999):

- Suporte a Visão Humana;
- Estabelecer uma base para pesquisas futuras;
- Criar um repositório de algoritmos básicos;
- Desenvolver uma plataforma para o desenvolvimento de produtos avançados;
- Suportar aplicações comerciais da tecnologia;
- Criar convenções para trabalhos futuros; e
- Criar uma comunidade autossustentável de usuários e desenvolvedores.

A arquitetura do ITK é baseada em fluxos de dados, isto é, os dados são representados por objetos de dados que são processados por objetos de processamento (filtros). Os objetos de dados e de processamento são conectados formando fluxos de processamento (*pipeline*). Fábricas de objetos são utilizadas para instanciar objetos, permitindo a extensão do sistema em tempo de execução.

A filosofia de implementação do ITK baseia-se no princípio da programação genérica. Por isso o ITK possui grande utilização de *templates*. Além do suporte gerado automaticamente

[1] US National Library of Medicine of the National Institutes of Health.

para as linguagens Tcl, Python e Java através da ferramenta *CableSwig* (INSIGHT..., 1999). E também é multiplataforma, usando o ambiente *CMake* (CROSS..., 2000) para configurar o processo de compilação em diferentes plataformas.

2.1.1.1 Arquitetura do ITK

A arquitetura do ITK permite ao programador focar nas partes mais importantes da implementação de filtros como a definição da API[2] da classe, a definição dos dados e os detalhes da própria implementação. Os conceitos básicos da programação em ITK são: (i) O Fluxo de Processamento de Dados, ou *pipeline*, é um grafo dirigido de processos e objetos de dados. O Fluxo recebe, opera e retorna um objeto de dados. (ii) O Filtro, ou objeto de processamento, é uma entidade que recebe um conjunto uma ou mais entradas de dados, realiza operações com eles e retorna um conjunto de uma ou mais saídas de dados. (iii) Objeto de Dados é a entidade que representa os dados em si e fornece acesso a eles. Um objeto de dados pode ser representado pelos tipos itk::Image ou itk::Mesh. (iv) Uma Região representa uma parte estruturada dos dados. (v) Uma Região de Malha, ou *Mesh*, representa uma região não estruturada.

Em ITK os filtros podem ser definidos de diversas formas variando na quantidade e tipos de entradas e saídas. Os principais tipos são: ImageToImageFilter recebe uma imagem e produz uma nova imagem. UnaryFunctorImageFilter é usado para definir um filtro que aplica uma função sobre uma imagem. BinaryFunctorImageFilter aplica uma função utilizando duas imagens para gerar uma nova. MeshToMeshFilter é um filtro que transforma malhas. LightObject é uma base abstrata para a criação de filtros que não se enquadram nas classes existentes.

O processo de execução de um fluxo de processamento realiza os seguintes passos:

1. Determinar quais filtros em um fluxo precisam ser executados. Esse passo otimiza o processo de execução do fluxo.

2. Inicializar os objetos de saída de dados preparando-os para novos dados. Esse passo também determina quanta memória será alocada e aloca a saída.

3. Determina a quantidade de dados que cada filtro deve processar para produzir uma saída de tamanho suficiente para os filtros subsequentes, levando em conta os limites de memória ou requisitos específicos de cada filtro.

[2] API, *Application Programming Inteface*, é o conjunto de rotinas e padrões definidos por um software para utilização de suas funcionalidades

4. Dividir os dados em pedaços de tamanho definido no passo anterior. Essa operação é chamada de *streaming* e é utilizada para o processamento em *multithreading* nativo do ITK e para o processamento de imagens que não cabem em memória.

5. Liberar a saída de dados dos filtros que não forem mais necessárias.

Para executar esses passos, o processo de execução negocia com cada um dos filtros que definem o fluxo e a quantidade necessária de dados para produzir uma saída determinada. Em última instância, o processo de negociação pode ser controlado pela requisição do usuário por um dado tamanho de imagem ou região (IBÁÑEZ et al., 2005).

2.2 PROCESSAMENTO PARALELO DE IMAGENS

Segundo Bräunl (2001), devido à grande quantidade de dados a serem processados em todos os níveis de processamento de imagens, o poder computacional necessário e o tempo de processamento são muito altos. Entretanto o processamento paralelo de uma imagem pode reduzir significativamente o tempo de processamento.

Para tornar o processamento paralelo de imagens mais eficiente possível, este deve ser realizado no nível dos pixels. Isso significa que cada pixel deve ser associado a um processador ou elemento de processamento. Então a mesma instrução é executada simultaneamente em cada pixel. A isso é dado o nome de paralelismo síncrono. Uma vez que os elementos de processamento executam as mesmas instruções simultaneamente não se faz necessária a sincronização entre os processos.

Existem três tipos de paralelismo: o paralelismo em nível de instrução (ILP), que encontramos nos processadores tradicionais de um núcleo; o processamento paralelo assíncrono; e o processamento paralelo síncrono, ou paralelismo de dados. A principal diferença entre o processamento paralelo assíncrono e síncrono consiste que no modelo assíncrono cada processador possui seu próprio fluxo de controle e executa suas próprias instruções, enquanto que no modelo síncrono todos os processadores recebem seus comandos de um controlador central.

Dessa forma, em sistemas de paralelismo síncrono, os processadores podem ser muito mais simples e, por isso, ocupam menos espaço no chip e podem ser integrados em uma densidade muito maior. O modelo de programação atribuído ao processamento paralelo assíncrono é chamado de *Multiple Instruction stream Multiple Data stream* (MIMD), pois diferentes fluxos de execução executam instruções simultaneamente em diferentes partes de dado. Enquanto que no modelo de programação atribuído ao processamento paralelo síncrono conhecido como *Sin-*

gle Instruction stream Multiple Data stream (SIMD), todos os fluxos de execução executam as mesmas instruções em diferentes partes de dado ou não executam nada. Além disso, o modelo SIMD permite integrar uma densidade muito maior de processadores, uma vez que eles são mais simples (BRÄUNL et al., 2001).

2.3 PROGRAMAÇÃO DE PROPÓSITO GERAL EM PLACAS GRÁFICAS

A exigência do mercado de processadores gráficos resultou na evolução das placas gráficas para se tornarem dispositivos altamente paralelos, com suporte a *multithreading*[3], com muitos processadores e com largo barramento de memória.

Devido à sua estrutura altamente paralela as placas gráficas estão deixando de ser dispositivos exclusivos para o processamento de aplicações gráficas, e começam a ser utilizadas para realizar processamento de propósito geral. A programação de propósito geral em placas gráficas permite aproveitar todo o poder de processamento de tais dispositivos, que pode possuir centenas de processadores independentes e diferentes tipos de memórias.

Os primeiros programas de propósito geral que aproveitavam o potencial das placas gráficas foram escritos através de APIs desenvolvidas exclusivamente para a computação gráfica, como é o caso das bibliotecas gráficas OpenGL[4] e Direct3D[5]. Porém o modelo de programação voltado para aplicações gráficas não se mostrou ideal à programação de propósito geral. Assim, foram desenvolvidos modelos de programação de propósito geral para permitir que o *hardware* das placas gráficas fosse utilizado. Neste contexto, em 2006, a NVidia lançou o Compute Unified Device Architecture (CUDA) com o objetivo de suprir a demanda do processamento de propósito geral em placas gráficas.

2.3.1 Compute Unified Device Architecture

O *Compute Unified Device Architecture*, ou CUDA, é a API desenvolvida pela NVidia que implementa o modelo de programação em placas gráficas. Através de um mecanismo de abstração do *hardware*, é possível proporcionar um ambiente simples de programação que utiliza bibliotecas de funções nas linguagens C e C++ e extensões. CUDA permite que o programador mantenha o foco na programação paralela ao simplificar o modelo de gerenciamento das *threads*. Além disso, essa abstração impede que atualizações do *hardware* façam programas deixarem de funcionar.

[3] Capacidade de executar vários processos simultaneamente
[4] http://www.opengl.org/
[5] http://en.wikipedia.org/wiki/Direct3D

O paralelismo exige que o programador pense no código de uma forma diferente da programação sequencial. Assim, o programador deve estar atento à arquitetura das placas gráficas e ao modelo de programação para começar a programar em CUDA. A série de artigos encontrada em (FARBER, 2008-2010), o guia oficial de programação em CUDA (NVIDIA..., 2010b) e o guia de instalação (NVIDIA..., 2010a) são boas referências literárias para os primeiros passos na programação em CUDA e para entender seus conceitos básicos. Os principais conceitos serão explicados a seguir.

CUDA implementa os modelos de paralelismo SIMT[6] e SIMD (NVIDIA..., 2010b) que através da arquitetura das placas gráficas gerenciam as *threads* e os dados, respectivamente. Com isso, pode-se dizer que CUDA implementa o paralelismo real em nível de *threads* com alto número de processadores e com uma arquitetura amplamente difundida. Além disso, o modelo CUDA fornece ao programador flexibilidade para modelar o paralelismo da forma mais eficiente (NICKOLLS et al., 2008). CUDA se mostra também um modelo especialmente adequado para resolver problemas de paralelismo de dados com alta intensidade aritmética[7].

O modelo de paralelismo implementado para CUDA permite ao programador fazer uso da arquitetura das placas gráficas de maneira simplificada. A arquitetura das placas gráficas pode possuir centenas de processadores além de regiões próprias de memória. Eles são divididos por multiprocessadores. Cada Multiprocessador recebe as instruções que devem ser executadas e utiliza um controlador para gerenciar a execução de seus processadores. Além disso, cada multiprocessador possui uma região de memória compartilhada entre seus processadores.

2.3.1.1 O Paradigma de Programação CUDA

O modelo CUDA compreende um conjunto extensões das linguagens C e C++ na qual o desenvolvedor escreve um programa serial que faz a chamada de *kernels*. *Kernel*, em CUDA, é uma função C que, quando invocada, é executada *N* vezes em paralelo por um conjunto de *N* *threads* CUDA na placa gráfica. Ou seja, todas as *threads* executam mesmo o código descrito pelo *Kernel* simultaneamente. As *threads* são organizadas hierarquicamente em *grids* de blocos de *threads*. Os blocos de *threads* são conjuntos de *threads* que podem cooperar através de barreiras de sincronização e acesso compartilhado a um espaço de memória privado para cada bloco. Os blocos organizam um conjunto de *threads* em até três dimensões. Um *grid* é um conjunto de blocos de *threads* que podem ser executados independentemente na mesma placa gráfica. Da mesma forma que os blocos, os *grids* organizam os blocos em conjuntos de até duas

[6] Single Intruction Multiple Thread

[7] Taxa de operações aritméticas em relação a taxa de operações de memória.

dimensões. Antes da chamada do *kernel*, o programador deve definir a dimensão dos blocos em *threads* e a dimensão do *grid* em blocos.

2.3.1.2 Modelo de Execução CUDA

A criação, o escalonamento e a finalização de todas as *threads* são controladas pelo sistema. De fato, placas gráficas realizam todo o gerenciamento de *threads* diretamente através do *hardware* (NICKOLLS et al., 2008). No modelo de execução em CUDA, as *threads* são executadas nos processadores de *threads* (ou *Stream Processors* - SP), os blocos de *threads* são executados nos multiprocessadores (MP), e os *grids* executam na placa gráfica.

Uma vez que as *threads* de um bloco compartilham o acesso a um espaço de memória e sincronizam através de barreiras, elas devem se situar no mesmo multiprocessador[8]. Porém o número de blocos pode exceder o número de multiprocessadores da mesma forma que o número de *threads* pode exceder o número de processadores. Dessa forma, blocos e *threads* executam de forma virtualizada.

O conceito de processadores virtuais é análogo ao conceito de memória virtual (BRÄUNL et al., 2001), pois eles não existem fisicamente, mas são escalonados nos multiprocessadores. A possibilidade de alocar mais blocos do que a quantidade de multiprocessadores permite flexibilidade para paralelizar o problema no nível mais conveniente de granularidade. A quantidade de blocos que podem ser executados concorrentemente depende da quantidade de recursos utilizados pelo *kernel*. Encontrar a combinação correta de recursos por bloco em um *kernel* significa manter o máximo de *threads* ativas e consequentemente obter o maior desempenho possível.

Cada bloco ativo é dividido em grupos de *threads* SIMD chamados de *Warps*. Cada *warp* contém o mesmo número de *threads*. A quantidade de *threads* em um *warp* é conhecida como tamanho do *warp* (*warp size*). Cada *warp* é executado nos multiprocessadores utilizando o modelo SIMD (FARBER, 2008-2010). Nas arquiteturas G80 e GT200, cada multiprocessador é composto de 8 processadores de *threads*, de forma que um multiprocessador é capaz de processar as 32 *threads* de um *warp* em 4 ciclos de *clock* (NVIDIA..., 2010b). Na arquitetura Fermi os multiprocessadores contam com 32 processadores permitindo o processamento de uma instrução de um *warp* em apenas um ciclo de *clock*. Todos os processadores de *threads* em um *warp* executam a mesma instrução simultaneamente, portanto a máxima eficiência é atingida quando todas as 32 *threads* de um *warp* possuem fluxos de execução iguais. Qualquer instrução de controle de fluxo (**if**, **switch**, **do**, **for**, **while**) pode afetar o tempo de execução ao

[8] Até a arquitetura GT200 um multiprocessador consiste em 8 núcleos de processamento. A partir da Arquitetura Fermi um multiprocessador possui 32 núcleos de processamento

dividir o *warp* em diferentes fluxos de execução. Quando isso acontece, o *warp* é dividido em grupos de *threads* com o mesmo fluxo de execução. Assim, a execução dos grupos de *threads* de um *warp* é serializada pelo multiprocessador, aumentando o número total de instruções para executar o *warp*. Os *warps* que possuem suas execuções particionadas são conhecidos como *warps* divergentes.

O modelo SIMD é eficiente e possui baixo custo do ponto de vista do *hardware*. Porém, as operações condicionais são serializadas, uma vez que o modelo exige que todas as *threads* executem as mesmas instruções ou não executem instrução nenhuma. Dessa forma, os comandos condicionais são executados em duas partes, primeiro são executadas todas as *threads* que avaliaram a condição como verdadeira, enquanto as demais *threads* permanecem inativas. Em seguida as *threads* que permaneceram inativas podem executar, se houver necessidade, enquanto as que executaram antes permanecem inativas (BRÄUNL et al., 2001). *Warps* são escalonados por tempo, ou seja, o escalonador periodicamente troca de um *warp* para outro para maximizar o uso dos recursos do multiprocessador (FARBER, 2008-2010).

Dado o modelo de execução em CUDA, certas configurações de execução podem desperdiçar processadores. Limitações de *hardware* podem impedir que os multiprocessador executem seus blocos utilizando todos os seus processadores. A ocupação da GPU mede a proporção de processadores ativos em GPU durante a execução de um *kernel*. Para maximizar a ocupação da GPU é preciso levar em conta, principalmente, os limites de *threads* por *warp*, *warps* por multiprocessador, registradores por multiprocessador e memória compartilhada por multiprocessador.

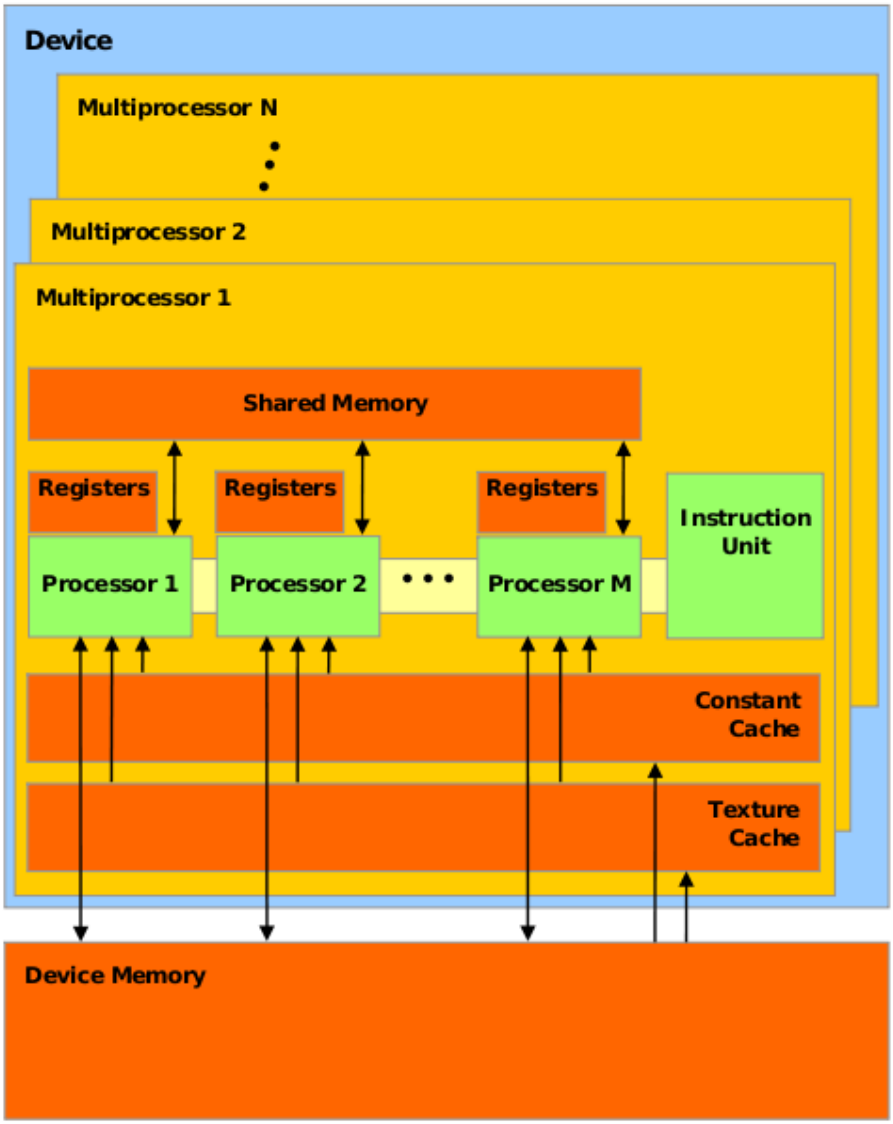
2.3.1.3 Modelo de Memória CUDA

As *threads* podem acessar os dados em diferentes espaços de memória durante suas execuções. Cada *thread* possui um conjunto de registradores locais. Os registradores são o tipo de memória de mais rápido acesso. Todas as *threads* de um bloco podem compartilhar o acesso a uma mesma memória compartilhada existente em cada multiprocessador. A memória compartilhada pode possuir velocidade de acesso semelhante aos registradores e tamanho de 16KB[9]. Todas as *threads* possuem acesso a uma mesma memória (DRAM) global. Este é o tipo de memória com o acesso mais lento na GPU, e possui algumas restrições para atingir o desempenho ótimo as quais serão tratadas a seguir.

Esses tipos de memória correspondem a espaços fisicamente separados. Além deles,

[9] Até a arquitetura GT200. Na arquitetura Fermi o tamanho da memória compartilhada foi expandido para até 64KB divididos com a cache L1.

Figura 2.1: MODELO DE MEMÓRIA.



Fonte: (NVIDIA. . . , 2010b)

outros tipos de memória são implementados através de abstrações da memória global. São eles, a memória local, a memória de constante, a memória de textura. Todas essas memórias estão representadas na Figura 2.1.

2.3.1.4 Memória Compartilhada

Quase tão rápida quanto os registradores, a memória compartilhada é dividida em módulos de memória de mesmo tamanho chamados de bancos de memória. Cada banco de memória possui 32 bits de largura, de forma que acessos a posições consecutivas a um vetor, por *threads* consecutivas são os mais eficientes. Conflitos de bancos ocorrem quando múltiplas requisições são feitas para dados no mesmo banco (ou ainda quando o mesmo endereço ou

múltiplos endereços requisitam acesso ao mesmo banco). Quando isso ocorre, o *hardware* serializa as operações de memória, o que força todas as *threads* a esperarem até que todas as requisições de memória estejam satisfeitas. Entretanto, se todas as *threads* leem de um mesmo endereço de memória, então um mecanismo de *broadcast* é utilizado automaticamente e a serialização é evitada. Dessa forma, ao evitar os conflitos de memória é possível fazer uso da memória compartilhada com desempenho muito próximo ao dos registradores.

2.3.1.5 Memória Global

A memória global utiliza a maior largura de banda apenas quando os acessos a ela são coalescentes[10] em um *half-warp*, que corresponde ao conjunto das *threads* da primeira ou segunda metade de um *warp*. O princípio da coalescência nos acessos à memória global define que, se todas as *threads* de um mesmo *half-warp* acessarem posições contíguas de memória, o paralelismo é garantido entre todas as *threads* envolvidas, caso contrário, os acessos à memória serão serializados. Assim o *hardware* pode armazenar ou recuperar dados em um menor número de transações.

Dispositivos CUDA da arquitetura G80 podem recuperar dados de 64 ou 128 bits em uma única transação. Se as transações de memória não puderem ser coalescentes, então uma transação de memória é realizada para cada *thread* no *half-warp*. Segundo o guia de Programação CUDA, os acessos não-coalescentes a dados de 32 bits são aproximadamente 10 vezes mais lentos que acessos coalescentes, 4 vezes mais lentos para dados de 64 bits e 2 vezes mais lentos para dados de 128 bits. Isso acontece pois nos acessos não-coalescentes os acessos de cada *thread* são serializados.

Dessa forma, acessos à memória global por todas as *threads* podem ser coalescentes quando as *threads* acessam tipos de dados de 32, 64 ou 128 bits. O endereço inicial e o alinhamento dos dados são importantes, uma vez que os dados acessados devem possuir o mesmo tamanho e as *threads* devem acessar posições contíguas. Assim a k-ésima *thread* deve acessar a k-ésima posição. Note, porém, que nem todas as *threads* em um *warp* precisam realizar acessos à memória para que o acesso seja coalescente. Isso é chamado de *warp* divergente (*divergent warp*).

Um multiprocessador gasta 4 ciclos para requisitar dados da memória para um *warp*. Acessar a memória global resulta em 400 a 600 ciclos adicionais de latência de memória (FAR-

[10] O termo original em inglês é *coalesced*. Em português o termo coalescência é um sinônimo de aglutinação. Assim, os acessos de todas as *threads* à memória são feitos de uma só vez em conjunto. "Aglutinando" vários acessos em um só, evitando sua serialização.

BER, 2008-2010). Isso significa uma diferença de 100 a 150 vezes no tempo de acesso entre a memória global e os registradores ou a memória compartilhada. A maioria da latência da memória global pode ser escondida definindo um grande número de blocos de *threads* e utilizando o máximo possível de registradores, memória compartilhada e memória de constante para realizar o acesso aos dados.

2.3.1.6 Memória Local

A memória local é uma abstração que implica em "local no escopo de cada *thread*". Por ser uma porção da memória global, seu desempenho é o mesmo de qualquer região da memória global. Normalmente, variáveis automaticamente alocadas residem nos registradores, porém o compilador pode optar por alocar variáveis automaticamente na memória global quando há muitas variáveis nos registradores, quando uma estrutura ocupar muito espaço nos registradores ou o compilador não puder determinar se o vetor é indexado com quantidade constante[11].

2.3.1.7 Memória de Constante

A cache de constante possui 64KB (8KB por multiprocessador), é compartilhada por todas as *threads* no modo somente-leitura (RO) e é otimizada em *hardware* especialmente para o caso onde todas as *threads* leem o mesmo endereço. A memória de constante possui latência de 1 ciclo quando não há falta de cache, mesmo pertencendo à memória global. Se as *threads* leem de diferentes localidades, o acesso é serializado, pois gera faltas de cache. O primeiro acesso à memória de constante geralmente não causa falta de cache devido a um dispositivo de *pre-fetch* (FARBER, 2008-2010). A memória de constante pode ser escrita apenas a partir da CPU e é persistente através de múltiplas chamadas de *kernels* na mesma aplicação.

2.3.1.8 Memória de Textura

A memória de textura é uma alternativa de caminho para o acesso à memória global que utiliza uma cache separada das demais memórias para otimizar o acesso. Seu acesso é somente-leitura a partir dos *kernels* e compartilhado por todas as *threads* de um *grid*. O uso da cache de textura possui pontos interessantes. Ela é otimizada para lidar com localidade espacial 2D. Sua cache é pequena, 8KB por multiprocessador, e pode fornecer seu melhor desempenho ao possuir todas as *threads* acessando posições de memória próximas (localidade espacial).

[11] Registradores não são endereçáveis, então um vetor deve ir para a memória local - mesmo se for um vetor de duas posições (um vetor pequeno) - quando o endereço do vetor não puder ser conhecido em tempo de compilação

Empacotar os dados também é uma técnica eficiente, uma vez que a leitura de textura de um **float4** é mais rápida do que 4 leituras de texturas de dados do tipo **float**.

O acesso à cache de textura é realizado através de referências de textura que são vinculadas à uma região de memória. Referências de textura distintas podem ser vinculadas às mesmas regiões de memória (utilizando sobreposição) ou à regiões diferentes. Cada unidade de textura possui uma memória interna que armazena dados da memória global, funcionando como uma cache (ou *buffer*). Por essa razão, a memória de textura pode ser utilizada como um mecanismo de relaxação do acesso das *threads* à memória global, devido aos exigentes requisitos de coalescência da memória global, pois estes requisitos não se aplicam à memória de textura.

Uma vez que o acesso otimizado aos dados é muito importante para o desempenho em GPU, o uso da memória de textura pode melhorar o desempenho em certas circunstâncias. O melhor desempenho é alcançado quando as *threads* de um *warp* acessam posições próximas na perspectiva da localidade espacial. CUDA fornece capacidades de busca (*fetch*) em 1D, 2D e 3D usando a cache de textura. Como a textura realiza uma operação de leitura de memória global apenas quando há uma falta na cache, é possível exceder o máximo teórico de largura da banda de memória global através do uso sensato da cache de textura. Isso faz da taxa de busca de textura (*texfetch*) uma métrica poderosa para analisar o desempenho de um *kernel* quando se utiliza texturas. Por exemplo, é possível fornecer aproximadamente 18 bilhões de buscas por segundo através da arquitetura G80 (FARBER, 2008-2010). O uso da cache de textura pode reduzir a penalidade por acessos não-coalescentes ou quase coalescentes[12].

Há outros benefícios da memória de textura. Dados empacotados podem ser distribuídos (via *broadcast*) a variáveis de diferentes *threads* através de uma única operação. E o *hardware* da unidade de textura é capaz de calcular a interpolação linear, bilinear ou trilinear sem ocupar os processadores de *threads*. Também, pode converter dados de entrada em 8 ou 16 bits para ponto flutuante de 32 bits com valores entre 0.0 e 1.0 ou -1.0 e 1.0.

2.3.1.9 Da Arquitetura G80 à Fermi

Em Novembro de 2006, a arquitetura G80 inaugurou o modelo de programação em GPU baseado em CUDA, incorporando as inovações ao modelo de computação em GPU que foram apresentadas até aqui. Entre as inovações introduzidas com o modelo G80 estão o uso de C para computação em GPU, a unificação dos processadores de vértice e pixel em um único tipo de processador capaz de processar vértices, pixels, geometria, e computação de propósito

[12] Por exemplo, endereço inicial desalinhado. Isso causa serialização no acesso à memória global.

geral (WHITEPAPER..., 2009).

Em Junho de 2008, melhorias na arquitetura G80 foram anunciadas criando assim a arquitetura GT200. As melhorias incluem o aumento na quantidade de processadores (de 128 para 240), o aumento da capacidade dos registradores de cada processador em duas vezes, a eficiência dos acessos à memória pela flexibilização da coalescência de memória e o suporte a ponto flutuante de precisão dupla.

A evolução da arquitetura GT200 veio recentemente com o anúncio da nova arquitetura Fermi. O foco no desenvolvimento da nova arquitetura foram: melhorar o desempenho de precisão dupla, proteger as aplicações de erros de memória, desenvolver uma arquitetura de cache, aumentar a memória compartilhada, otimizar a troca de contexto entre aplicações e otimizar operações atômicas. Seguindo esse foco, as principais melhorias da arquitetura Fermi incluem (WHITEPAPER..., 2009):

- Uso de até 512 processadores CUDA (CUDA *cores*) através do aumento da quantidade de processadores por multiprocessador de 8 para 32;
- Duplo escalonador de *Warps* permite escalonar e disparar instruções simultaneamente para dois *warps*;
- A melhoria de desempenho de aritmética de ponto flutuante em precisão dupla em até 8x;
- NVidia *Parallel Data Cache* fornece uma hierarquia de cache para o acesso a memória. Os 64KB da memória compartilhada podem ser divididos entre a própria memória e uma cache L1. Enquanto que uma cache L2 otimiza o acesso à memória global. Essa abordagem permite reduzir a latência no acesso à memória;
- A tecnologia NVidia *Giga Thread* permite a execução de multiplos kernels simultaneamente em uma mesma GPU e transferências de dados bidirecionais simultâneas entre CPU e GPU;
- *Error Check and Correction*, ou ECC, permite a detecção e correção de erros de software que possam acontecer no armazenamento dos dados em qualquer das memórias da GPU.

2.4 PROCESSAMENTO DE IMAGENS MÉDICAS USANDO CUDA

Segundo Nickolls (2008), há um número considerável de aplicações desenvolvidas em CUDA, apesar do pouco tempo desde sua introdução. No processamento de imagens da área

médica podemos destacar aplicações como a reconstrução de imagens de ressonância magnética (MRI) e a classificação de imagens de estroma.

Em placas gráficas com a arquitetura GT200 (NVIDIA..., 2010b), a reconstrução de MRI alcançou ganhos de desempenho, se comparado a implementações tradicionais em CPU, sendo até 263 vezes mais rápida nos testes realizados para a implementação em (STONE et al., 2007). Da mesma forma, no algoritmo de classificação de estroma encontrado em (HARTLEY et al., 2008) foi alcançada uma grande redução no tempo de processamento do algoritmo para uma imagem de 109110×80828 pixels, que era de aproximadamente 11 horas e 39 minutos usando uma implementação simples em MATLAB e passou a ser de cerca de 1 minuto e 47 segundos utilizando uma solução combinando uma grade computacional e CUDA.

2.5 DETECTOR DE BORDAS CANNY

A implementação do detector de bordas Canny utilizando o modelo de programação CUDA foi proposto neste trabalho como um exemplo de aplicação para a utilização de CUDA em fluxos ITK.

Figura 2.2: EXEMPLO DE SAÍDA DO DETECTOR DE BORDAS CANNY.

(a) Original (b) Bordas

Fonte: Autor.

O esquema de detecção de bordas proposto por Canny (1986) aceita imagens digitais discretas como entrada e produz um mapa de bordas como saída. A Figura 2.2 mostra um exemplo de execução do esquema de detecção de bordas Canny. O mapa de bordas pode incluir informações explícitas sobre a posição, a magnitude e orientação dos pixels de borda. Entretanto muitas implementações utilizam como resultado de saída apenas a posição das bordas, retornando uma imagem binária onde os valores podem representar apenas o fundo ou uma

borda em destaque. Canny (1983) ainda define os 3 critérios de desempenho que a detecção ótima de uma borda deve possuir:

(1) Boa detecção: Deve haver baixa probabilidade de bordas reais não serem marcadas como pontos de bordas, e de marcar erroneamente pontos que não são bordas. Uma vez que essas duas probabilidades são funções monoliticamente decrescentes do sinal de saída em relação à taxa de ruído, este critério corresponde a maximizar a magnitude do sinal em relação à taxa de ruído.

(2) Boa localização: Os pontos marcados como bordas pelo operador deve ser tão próximo quanto possível do centro da borda real.

(3) Apenas uma resposta para uma borda: Isso é implicitamente definido em (1) uma vez que dois resultados próximos correspondem à mesma borda, um deles deve ser considerada uma borda falsa. Entretanto, a forma matemática do primeiro critério não define o requisito de múltiplas respostas e isso deve ser colocado explicitamente.

Dessa forma, o autor definiu que a busca ótima de uma borda deve obedecer a um conjunto de critérios que maximizam a probabilidade de detectar bordas verdadeiras enquanto minimizam a probabilidade de bordas falsas. A partir dessas definições, ele descobriu que o cruzamento em zero (*zero-crossing*) da segunda derivada de uma imagem suavizada possui uma medida razoável de bordas verdadeiras encontradas. E o algoritmo de Canny possui os seguintes passos:

1. Suavização da imagem de entrada;
2. Cálculo do gradiente da imagem suavizada;
3. Supressão dos valores não máximos de gradiente em sua direção; e
4. Limiarização dos picos de gradiente para eliminar bordas falsas, muitas vezes geradas por ruído na imagem.

Para suavizar a imagem, o detector de bordas Canny utiliza a convolução Gaussiana. A suavização da imagem de entrada tem como objetivo reduzir a taxa de ruído da imagem a fim de gerar menos bordas falsas. Porém vale destacar que a suavização reduz os valores de gradiente nos passos seguintes podendo resultar na perda de bordas reais.

Em seguida a imagem é convolucionada com um operador de primeira ou segunda derivada para determinar regiões de altas mudanças de intensidade. A magnitude e direção do

gradiente em cada pixel é calculado neste passo. Os pixels onde se encontram os valores de máximo e mínimo (ou de cruzamento em zero) são os pontos de interesse da imagem, pois representam as áreas de maior mudança de intensidade da imagem. Os picos de gradiente[13] representam o conjunto de possíveis bordas. Para ignorar os valores que não sejam considerados picos de gradiente, aplica-se a técnica de supressão não-máxima (*non-maximum supression*). Esse passo visa garantir que apenas os pontos de interesse da imagem sejam mantidos.

Por fim, uma técnica de dupla limiarização conhecida como histerese é aplicada ao longo das possíveis bordas para determinar o conjunto final de bordas que formará o mapa de bordas. Segundo Canny (1983), quase todos os esquemas de detecção de bordas usam algum tipo de limiarização. Se os limiares não são fixos *a priori*, mas determinados de alguma forma pelo algoritmo, diz-se que o detector emprega limiarização adaptativa.

2.5.1 Implementações do Detector de Bordas Canny usando CUDA

Em (LUO; DURAISWAMI, 2008) encontra-se a implementação em CUDA do algoritmo Canny para a biblioteca de processamento de imagens OpenVIDIA (FUNG; MANN, 2005). Em comparação com o filtro desenvolvido para a biblioteca OpenCV que utiliza o *hardware* especializado para o processamento de multimídia SSE, a implementação em CUDA foi até 3,87 vezes mais eficiente, e em comparação com a versão simples em MATLAB o desempenho chegou a ser aproximadamente 100 vezes mais rápido (LUO; DURAISWAMI, 2008).

A eficiência das placas gráficas na implementação do algoritmo Canny também foi utilizada na aplicação do detector de bordas tempo real nos *frames* de um vídeo (GÓMEZ-LUNA et al., 2009). Seus testes foram executados para 500 frames de 4 vídeos com resolução de 352×288 pixels. Em uma comparação dos melhores resultados entre essa aplicação e outra implementação usando 4 *threads* (em um processador com 4 núcleos) em OpenMP, ela foi de 2,8 a 3,5 vezes mais rápida. A versão do algoritmo em OpenMP usando uma CPU Intel®Xeon®de 8 núcleos teve desempenho semelhante a aplicação em CUDA para vídeos com mais pontos de contorno (bordas), mas tendo desempenho em média 22% inferior para os outros vídeos testados. Porém o preço da placa gráfica utilizada (GeForce GTX 280) corresponde a apenas 10% do preço da CPU de 8 núcleos (GÓMEZ-LUNA et al., 2009).

[13] Os picos de gradiente são os pixels que possuem valores de magnitude máximos locais em suas respectivas direções.

2.6 INTEGRANDO FILTROS IMPLEMENTADOS USANDO CUDA EM FLUXOS DO ITK

Na tentativa de aproveitar o paralelismo das placas gráficas para melhorar o desempenho de diversos filtros de processamento de imagens, a integração entre filtros implementados em CUDA e fluxos de processamento de imagens em ITK está sendo discutida na comunidade ITK, inclusive com a formalização da proposta de integração do ITK com a programação em placas gráficas (PROPOSALS..., 2007-2009) e a discussão de um *framework* para permitir o uso de GPUs na versão 4 do ITK (ITK..., 2010-2011). Portanto, a comunidade ITK acredita que o uso de placas gráficas oferece a oportunidade de acelerar a execução de alguns filtros do ITK.

A metodologia sugerida é utilizar o mecanismo de fatoração para criar versões especializadas de filtros de imagem específicos que utilizem implementações baseadas no uso de placas gráficas. Essas versões seriam criadas apenas para alguns algoritmos considerados críticos para o benefício da comunidade ITK. Para isso o uso da linguagem OpenCL é indicado (PROPOSALS..., 2007-2009) por possuir uma abordagem genérica para a criação de código paralelo para o ITK e por ser melhor para a programação multi-GPU assíncrona. Ainda há divergências sobre esse ponto uma vez que há também vantagens no uso de CUDA. Segundo Joe Steam da NVidia em uma discussão[14] sobre a versão 4 do ITK, CUDA possui uma vasta coleção de bibliotecas e é mais fácil para otimizar desempenho. É necessário, também, repensar o mecanismo *multithreading* atual do ITK que é baseado em POSIX *threads* e SGI (IBÁÑEZ et al., 2003) para que ele suporte o paralelismo das placas gráficas. Até o momento, não há nenhuma implementação oficial adotada pela comunidade ITK de como programar filtros de processamento de imagens em ITK usando CUDA, de forma generalizada e eficiente. E continuam as discussões sobre o suporte ao processamento em GPU que deve ser incluído na versão 4 do ITK.

Nesse sentido, algumas tentativas de utilizar CUDA na criação de funções e filtros para execução em fluxos de processamento de imagens do ITK foram desenvolvidas. Como por exemplo a implementação do registro de imagens em CUDA para o ITK (ITK..., 2008). Essa implementação é atribuída à *National Aliance for Medical Image Computing* (NA-MIC), mas não há código disponível, apenas uma descrição do *framework* proposto. Segundo essa descrição, o *framework* consiste na implementação em CUDA de uma classe para o acesso à memória em CUDA, e das classes responsáveis pela métrica, transformada e interpolação. Uma das primeiras tentativas de integrar filtros de processamento de imagens implementados

[14] Disponível em http://www.itk.org/Wiki/ITK_Release_4/GPU_Acceleration/Tcon-2010-11-22

em CUDA com o ITK de forma genérica foi o projeto CUDAITK (JEONG, 2007).

2.6.1 CUDAITK

A proposta do projeto CUDAITK é criar implementações de filtros do ITK utilizando a linguagem CUDA com o intuito de aproveitar o paralelismo das placas gráficas e melhorar o desempenho dessas funções. Outra motivação para investir esforços nesse sentido é que o grau de paralelismo atual do ITK é limitado pelo número de CPUs. O foco do projeto é o processamento de volumes tridimensionais, pois a maioria dos conjuntos de dados de ressonância magnética e tomografia são volumes 3D (JEONG, 2007).

Na implementação do CUDAITK[15], o código CUDA é integrado ao ITK de forma transparente aos usuários, eliminando a necessidade de alterar o código de programas em ITK já existentes, exigindo apenas a atribuição de uma variável de ambiente para executar a versão original ou a versão em CUDA do filtro utilizado. A versão do ITK utilizada para desenvolver o CUDAITK foi a versão 3.12.0.

Foram implementados em CUDA os filtros da média, mediana, gaussiano, e difusão anisotrópica. Segundo Jeong (2007), as implementações em CUDA desses filtros tiveram os seguintes desempenhos: os filtros de convolução da média e gaussiano ficaram aproximadamente 140 e 60 vezes mais rápidos que suas versões originais em ITK, respectivamente. O filtro da mediana que utilizou o método da mediana por bisseção de histograma, ficou aproximadamente 25 vezes mais rápido. E o filtro de difusão anisotrópica foi aproximadamente 70 vezes mais rápido.

Os trabalhos futuros propostos no CUDAITK incluem a redução da transferência de dados entre memórias (devido ao alto custo de transferir e recuperar os dados na memória das placas gráficas); suporte a *pipelining*; uma interface nativa para a programação em placas gráficas; implementação independente de plataforma (que não dependa do *hardware*. OpenCL (OPENCL, 2008) é vista como uma solução); entre outros. Uma outra solução, porém, resolveu a maioria desses problemas utilizando uma abordagem diferente. Esta solução é conhecida por CITK.

[15] O código da implementação do CUDAITK está disponível no repositório do souceforge em http://sourceforge.net/projects/cudaitk/.

2.6.2 CITK

O *Cuda Insight Toolkit*, ou CITK, (BEARE et al., 2011) foi criado com o propósito de estender o suporte do ITK às arquiteturas de computação de propósito geral em placas gráficas. É um projeto de código aberto com licensa BSD que tem atraído o interesse inclusive de algumas Universidades Australianas em continuar o desenvolvimento. O código fonte para utilização está disponível em (CUDA..., 2010). O CITK começou como uma pesquisa de um curso de verão do VPAC (*Victorian Partnership for Advanced Computing*). O VPAC é uma instituição sem fins lucrativos formada por um consórcio de Universidades Australianas. Segundo os desenvolvedores do CITK, após a conclusão do projeto eles entraram em contato com os desenvolvedores do ITK que mostraram interesse no projeto.

A abordagem utilizada para implementar o CITK consiste em alterar levemente a arquitetura do ITK para suportar CUDA. Para isso foi necessário alterar a classe que define o conceito de *pixel container*[16] que é utilizada para definir um objeto imagem na classe itkImage. Assim uma nova classe chamada CudaImportImageContainer foi criada para estender a funcionalidade da ImportImageContainer.

A nova classe permite completa compatibilidade com as componentes ITK existentes. Ela é usada para gerenciar os dados da imagem nas memórias da GPU e da CPU. Quando um filtro requer os dados de uma imagem, a classe verifica em qual memória estão os dados mais recentes referente àquela imagem. Caso os dados mais recentes estejam na memória do processador (CPU ou GPU) que será utilizado, a classe apenas retorna o ponteiro para os dados correto. Caso contrário, a classe CudaImportImageContainer efetua uma cópia dos dados para a memória do processador que será utilizado antes de retornar o ponteiro correto. Essa abordagem possui as vantagens de reduzir a quantidade de cópias de dados entre memórias ao mínimo necessário, permitir o encadeamento de filtros independente da implementação de cada um, permitir a programação de forma nativa ao ITK e a utilização mista de filtros implementados para executar em CPU e GPU.

[16] Pixel Container é uma estrutura do ITK utilizada para esconder e manipular os dados da imagem do usuário.

3 DESENVOLVIMENTO

Neste capítulo encontra-se a descrição das seguintes implementações desenvolvidas[1] neste trabalho: (i) a implementação em CUDA dos detectores de bordas Canny (CudaCanny) e Sobel (CudaSobel) para a biblioteca ITK utilizando a extensão CITK, (ii) uma classe para armazenar configurações CUDA para o CITK (itkCudaInterface), e (iii) 4 algoritmos para avaliar técnicas de otimização em CUDA. A presente implementação dos filtros CudaCanny e CudaSobel é capaz de processar apenas imagens em duas dimensões.

3.1 DETECTOR DE BORDAS CANNY IMPLEMENTADO EM CUDA PARA O ITK

A implementação de um filtro de processamento de imagens usando CUDA para o ITK permite identificar as maiores dificuldades envolvidas na integração de filtros implementados em CUDA com os fluxos de processamento do ITK. Dessa forma escolhemos o detector de bordas Canny para ser implementado em CUDA para o ITK por ser um filtro composto (implementado através de uma combinação de filtros) que permite avaliar a implementação de um fluxo de imagens através de diversos filtros. Além disso o Canny é um algoritmo que apresenta partes facilmente paralelizáveis e outras intrinsecamente sequenciais. A implementação em CUDA do filtro de detecção de bordas Canny foi chamado de CudaCanny. A abordagem utilizada na implementação do CudaCanny foi a mesma utilizada no Canny disponível no ITK baseada no artigo (CANNY, 1986). O Algoritmo 3.1 descreve os passos básicos do algoritmo Canny.

O primeiro passo da implementação foi a definição da classe que iria comportar o filtro. Seguindo as indicações do Guia do ITK (IBÁÑEZ et al., 2005), a maneira mais simples de criar um novo filtro é utilizar a classe de um filtro existente que possua características semelhantes às do filtro que se deseja implementar. Assim a base para a implementação foi a classe itkCannyEdgeDetectionImageFilter do ITK. A nova classe criada para o CudaCanny foi chamada de itkCudaCannyEdgeDetectionImageFilter. Por ser um filtro composto, grande parte

[1] Os códigos das implementações desenvolvidos encontram-se disponíveis em http://www.inf.ufpr.br/vri/alumni/2011-LuisLourenco/

ALGORITMO 1 DETECTOR DE BORDAS CANNY

Suavização da imagem de entrada com o filtro Gaussiano;
Cálculo da segunda derivada direcional da imagem suavizada;
Non-Maximum Supression: o Zero-Crossing da segunda derivada é encontrado e o sinal da terceira derivada direcional é usada para encontrar os extremos corretos;
Aplicação da Histerese sobre magnitude do gradiente (multiplicada com o Zero-Crossing) da imagem suavizada para encontrar e ligar as bordas;

do algoritmo utiliza outros filtros em sua execução, como a Gaussiana e o *Zero Crossing*. No entanto os métodos que calculam as Derivadas Direcionais e a Histerese foram reimplementados na própria classe para serem executadas em placa gráfica. Segue a explicação de cada uma das classes e métodos implementados em CUDA para o CudaCanny.

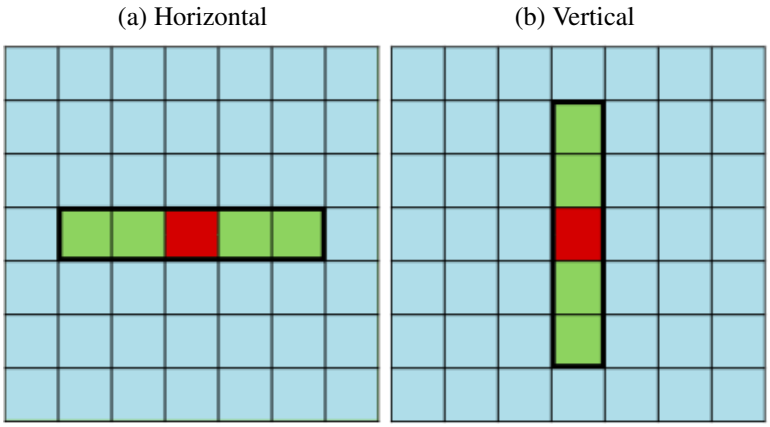
3.1.1 Suavização Gaussiana

A Suavização Gaussiana é o primeiro passo do detector de bordas Canny. Seu objetivo é diminuir o ruído da imagem para que os passos seguintes detectem menos bordas falsas. Ela foi implementada em uma nova classe chamada itkCudaDiscreteGaussianImageFilter. Além da imagem de entrada, o filtro Gaussiano recebe a variância que representa o desvio padrão (σ) que determina a largura da máscara gaussiana.

Para o cálculo das máscaras gaussianas foi utilizada a classe do ITK itkGaussianOperator. A convolução é realizada por uma classe implementada em CUDA chamada itkCuda2DSeparableConvolutionImageFilter. A classe implementa a convolução separável de duas máscaras unidimensionais quaisquer em uma imagem bidimensional. A convolução foi desenvolvida utilizando o princípio da separabilidade da máscara (BRÄUNL et al., 2001). Dessa forma são realizadas duas convoluções unidimensionais ao invés de uma bidimensional. As duas convoluções unidimensionais, além de mais simples, utilizam menos instruções, pois a convolução de um operador de tamanho *MxM* pode ser executada em apenas $2 * (M - 1)$ operações ao invés de $M^2 - 1$ (BRÄUNL et al., 2001). A convolução de cada uma das máscaras unidimensionais é realizada em sequência, de forma que a imagem resultante da primeira convolução é a entrada da segunda, e o resultado da segunda convolução já corresponde à suavização da imagem de entrada do filtro.

A convolução foi implementada da seguinte forma: A primeira máscara e a imagem de entrada são vinculadas (*bind*) à referências de textura (utilizada para indexar uma porção da memória global que fará uso do cache de textura). Assim que o *kernel* de cada convolução é invocado, a máscara gaussiana é carregada na memória compartilhada através de acessos à cache de textura. Em seguida o *kernel* entra no laço de execução que realiza a convolução. Cada

Figura 3.1: CONVOLUÇÃO GAUSSIANA.



Fonte: Autor.

thread calcula o valor de um pixel da imagem utilizando os valores de determinados vizinhos, como ilustra a Figura 3.1. Na Figura 3.1 os pixels em vermelho e verde representam os pixels utilizados em uma convolução, por exemplo, para calcular o valor do pixel em vermelho com uma máscara de 5 pixels de largura. Os demais pixels da Figura representam uma porção da própria imagem. Dessa forma, dada uma imagem de entrada I, uma máscara M e uma imagem convolucionada C, as convoluções horizontal e vertical podem ser definidas respectivamente pelas seguintes expressões:

$$C[i] = \sum_{j=-h}^{h} I[i+j] * M[j] \tag{3.1}$$

$$C[i] = \sum_{j=-h}^{h} I[i+(j*w)] * M[j] \tag{3.2}$$

Onde h corresponde à metade do tamanho da máscara e w é a largura da imagem. Então para cada pixel $I[i]$, a Figura 3.1 mostra os pixels (em verde e vermelho) que são acessados para calcular o valor de $C[i]$. O resultado da convolução para cada *thread* atribui um novo valor ao pixel $C[i]$ correspondente.

3.1.2 Detecção de Bordas baseada em Geometria Diferencial

Um meio de identificar pontos de borda em uma imagem é calcular os máximos da magnitude do gradiente da imagem, ou seja, os pontos onde ocorrem mudanças significativas de tonalidade. O gradiente da imagem pode ser adquirido através de um operador de primeira ou segunda derivadas. Segundo Canny (1983), um operador direcional mostra melhores resultados que um operador não direcional por possuir melhor relação entre as taxas de sinal e ruído (SNR).

Os pontos de cruzamento em zero (*zero crossing*) da segunda derivada direcional servem para localizar os valores máximos do gradiente.

Sendo assim, os desenvolvedores do ITK implementaram métodos para encontrar os pixels de borda baseados em Geometria Diferencial (LINDEBERG, 1998; KIDWAI; SIBAI; RABIE, 2009). Através do sistema de coordenadas curvilíneas (u,v) na qual para cada ponto a direção v é paralela à direção do gradiente da imagem suavizada (L), e para cada ponto a direção u é perpendicular. Para cada ponto $P = (x,y) \in \mathbb{R}^2$, ∂_v denota o operador da derivada direcional na direção do gradiente de L e ∂_u a derivada direcional perpendicular. Então para qualquer $P \in \mathbb{R}^2$ a magnitude do gradiente é definida pela convolução $\partial_v * L$, denotado como L_v. A magnitude do gradiente, ou derivada direcional em v, pode ser calculada através das derivadas direcionais em x e y utilizando os seguinte operadores:

$$\partial_x = \begin{pmatrix} -1/2 & 0 & 1/2 \end{pmatrix} \tag{3.3}$$

$$\partial_y = \begin{pmatrix} 1/2 \\ 0 \\ -1/2 \end{pmatrix} \tag{3.4}$$

Ao convolucionar a imagem suavizada L utilizando esses dois operadores é possível obter as derivadas direcionais L_x e L_y. Essas derivadas direcionais podem ser utilizadas para obter a derivada direcional em v que define a magnitude do gradiente de L através da seguinte fórmula:

$$L_v = \sqrt{L_x^2 + L_y^2} \tag{3.5}$$

Ao assumir as derivadas direcionais de segunda e terceira ordens de L na direção v, L_{vv} e L_{vvv}, pode-se obter os pixels de L que serão definidos como possíveis pixels de borda quando (LINDEBERG, 1998; KIDWAI; SIBAI; RABIE, 2009):

$$\begin{cases} L_{vv} = 0 \\ L_{vvv} \leq 0 \end{cases} \tag{3.6}$$

As seguintes derivadas serão utilizadas para definir as derivadas de segunda e terceira ordem de L em v:

$$L_{vvx} = L_{vv} * \partial_x \tag{3.7}$$

$$L_{vvy} = L_{vv} * \partial_y \tag{3.8}$$

L_{xx}, L_{yy} e L_{xy} são definidos pela convolução de L com os seguintes operadores:

$$\partial_{xy} = \begin{pmatrix} 1 & -2 & 1 \end{pmatrix} \tag{3.9}$$

$$\partial_{xx} = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix} \tag{3.10}$$

$$\partial_{yy} = \begin{pmatrix} -1/4 & 0 & 1/4 \\ 0 & 0 & 0 \\ 1/4 & 0 & -1/4 \end{pmatrix} \tag{3.11}$$

Em termos de derivadas em coordenadas cartesianas podemos calcular as derivadas direcionais de segunda e terceira ordem através das seguintes equações[2]:

$$L_{vv} = \frac{(L_x^2 L_{xx}) + 2L_x L_y L_{xy} + (L_y^2 L_{yy})}{L_v^2} \tag{3.12}$$

$$L_{vvv} = L_{vvx}\left(\frac{L_x}{L_v}\right) + L_{vvy}\left(\frac{L_y}{L_v}\right) \tag{3.13}$$

A função que calcula a segunda derivada recebe a imagem L e a vincula a uma referência de textura. Em seguida o *kernel* da segunda derivada é executado. Cada *thread* do *kernel* começa armazenando os vizinhos (vizinhança-8) de seu pixel em memória compartilhada, pois esses valores serão utilizados mais de uma vez, e um acesso de textura e dois de memória compartilhada é mais eficiente que dois acessos à cache de textura. Com todos os vizinhos do pixel em memória compartilhada, a convolução para obter as derivadas L_x, L_y, L_{xx}, L_{yy} e L_{xy} pode ser calculada através de multiplicações e somas dos valores dos vizinhos sem o uso de laços de execução. Ao final do *kernel* as *threads* armazenam seus respectivos valores de L_{vv} em uma nova posição de memória alocada para esse fim.

[2] A implementação dessas equações pode ser encontrada no código do Canny do ITK.

A função que calcula a terceira derivada funciona de forma semelhante. Ela recebe a imagem L e a segunda derivada L_{vv}, as vincula a referências de textura e inicia o *kernel* que vai calcular a terceira derivada. Cada *thread* do *kernel* primeiro armazena os vizinhos (vizinhança-4) referentes ao pixel em L e em L_{vv} em memória compartilhada. Em seguida calcula os valores de L_x, L_y, L_{vvx} e L_{vvy}. Com esses valores é possível calcular a magnitude do gradiente (L_v) e a terceira derivada de L (L_{vvv}).

3.1.3 Non-Maximum Supression

Até aqui o algoritmo do Canny produziu a segunda derivada da imagem suavizada (L_{vv}) e a magnitude do gradiente nos pontos onde a terceira derivada da imagem suavizada (L_{vvv}) é menor ou igual a zero. No entanto, dada a definição da localização dos pixels de borda da Equação 3.6, ainda falta um passo para definir os possíveis pontos de borda, que é identificar os pontos onde a segunda derivada é zero. Esse passo é conhecido por *Non-Maximum Supression* (NMS), uma vez que os pontos onde a segunda derivada de L em v possui valor nulo correspondem aos pontos de máximo da magnitude do gradiente.

Para esse passo então é utilizado o filtro *Zero Crossing*. Esse filtro realiza a comparação de cada pixel com seus vizinhos buscando por cruzamentos em zero, ou seja, mudanças de sinal ou valores nulos seguidos de valores não nulos. Note que valores nulos seguidos não configuram um cruzamento em zero. Dessa forma o filtro foi implementado vinculando a L_{vv} a uma referência de textura. Cada *thread* do *kernel* da NMS carrega um pixel $L[i]$ e seus vizinhos (vizinhança-4) para seus registradores. E em seguida faz a comparação do pixel com seus vizinhos. Caso o pixel $L[i]$ e pelo menos um de seus vizinhos possuam sinais diferentes ou um nulo e o outro não-nulo, a posição i recebe 1 como resultado, indicando ser pixel de borda. Caso contrário recebe valor 0. O resultado do filtro *Zero Crossing* é uma imagem binária das posições da segunda derivada da imagem de entrada suavizada onde ocorrem cruzamentos em zero.

3.1.4 Histerese

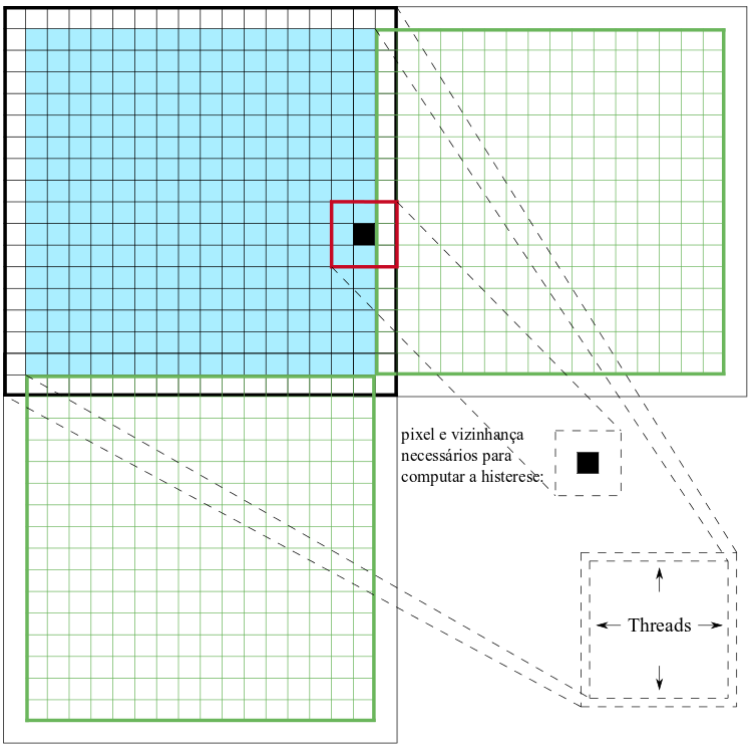
A última parte do Canny é a histerese que consiste em ligar as bordas relevantes da imagem. O primeiro passo da histerese consiste em multiplicar os valores da imagem contendo a localização das bordas (0 ou 1) com a magnitude do gradiente, obtendo os pontos de máximo da magnitude do gradiente, e realizar uma dupla limiarização no resultado classificando os pixels em bordas definitivas (*BD*), possíveis (*BP*) ou não-bordas (*NB*).

O passo seguinte envolve a busca de vizinhos das bordas possíveis (*BP*) que sejam bordas definitivas (*BD*). Neste caso, as *BP* tornam-se também *BD*. O efeito desse passo é que as linhas de bordas que ficaram entre os limiares superior e inferior na dupla limiarização serão ligadas às *BD* da imagem se estiverem próximas a outras *BD*. Por fim as *BP* que não foram reclassificadas como *BD* são reclassificadas como não-bordas (*NB*). Isso gera uma imagem apenas com *BD* e *NB* como resultado final.

A função da histerese vincula os dados recebidos a referências de memória. Para cada pixel $p[i]$ e limiares superior T_h e inferior T_l, a dupla limiarização é calculada da seguinte forma:

$$\begin{cases} BD, & \text{se } p[i] > T_h \\ BP, & \text{se } T_h \geq p[i] > T_l \\ NB, & \text{se } p[i] \leq T_l \end{cases} \tag{3.14}$$

Figura 3.2: SEGMENTAÇÃO DE IMAGEM EM REGIÕES.



Adaptado de (HARTLEY et al., 2008).

A classificação dos pixels no *kernel* de preparação define a imagem de histerese, que será usada no *kernel* que executa a histerese. Para o CudaCanny a estratégia tradicional da histerese que utiliza uma fila de bordas definitivas que segue ligando as bordas possíveis vizinhas a cada borda definitiva não se mostrou eficiente, pois essa estratégia pode ser executada apenas sequencialmente para um pixel de cada vez. Além disso, muitas vezes essa operação depende

de acessos à toda a imagem.

A estratégia utilizada no CudaCanny foi baseada no algoritmo desenvolvido em (LUO; DURAISWAMI, 2008). O *kernel* da histerese, definido pelo Algoritmo 3, utiliza um vetor de 324 posições para armazenar uma região de 18x18 pixels da imagem de entrada em memória compartilhada que é preenchida pelas *threads* de um bloco de 256 *threads*. O centro da região contém 16x16 pixels da imagem mais uma borda extra de 1 pixel que sobrepõe as regiões vizinhas da imagem, como mostra a Figura 3.2, de forma que todos os pixels possam acessar o valor de seus vizinhos. Cada *thread* preenche a memória compartilhada com um pixel da imagem. As extremidades de cada região são preenchidas pelas *threads* mais próximas. No caso de as extremidades extrapolarem a imagem, seus valores são definidos como não-bordas.

Assim que as regiões da imagem são carregadas na memória compartilhada, cada pixel classificado como borda possível busca seus vizinhos. Se dentre eles houver algum que seja borda definitiva, então o pixel é reclassificado como borda definitiva. Essa operação é executada em um laço até que nenhum pixel tenha sua classificação alterada. Essa parte da execução é conhecida como laço interno e é responsável por ligar as bordas definitivas dentro de cada bloco. Ao final do *kernel* os blocos que tiveram pixels modificados atualizam os valores de seus pixels na imagem de histerese em memória global. Esse *kernel* é executado em um laço externo até que não hajam modificações em toda a imagem de histerese, como mostram os Algoritmos 2 e 3. Caso algum pixel durante a execução desse *kernel* seja modificado, o *kernel* deve ser executado novamente.

ALGORITMO 2 HISTERESE

```
Recebe a localização das bordas e magnitude do gradiente
Executa o Kernel de Preparação da Histerese
repeat
    Executa Kernel da Histerese
until Nenhum pixel tenha sido modificado
Executa o Kernel de Finalização da Histerese
Retorna a imagem com as bordas
```

Por fim, a função da histerese vincula o resultado do *kernel* da histerese a uma referência de textura para a finalização da histerese. Cada *thread* do *kernel* de finalização recebe um pixel e anula os valores dos pixels que não são de borda definitiva. Assim o resultado da histerese contém apenas bordas definitivas e não-bordas.

ALGORITMO 3 KERNEL DA HISTERESE

```
Carrega uma região da imagem de tamanho 18x18 para a memória compartilhada de cada
multiprocessador
modified ← false
repeat
   modified_region ← false
   Sincroniza as threads do mesmo bloco
   if Pixel = BP e ∀ pixel vizinho = BD then
      Pixel ← BD
      modified ← true
      modified_region ← true
   end if
   Sincroniza as threads do mesmo bloco
until modified_region = false
if modified = true then
   Atualiza regiões modificadas na imagem
end if
Soma modified de todos os blocos
Armazena a soma em variável na memória global
```

3.2 DETECTOR DE BORDAS SOBEL IMPLEMENTADO EM CUDA PARA O ITK

Uma das primeiras implementações do CudaCanny utilizou o detector de bordas Sobel para calcular o magnitude e a direção da gradiente. Mais tarde com a mudança de abordagem para a detecção de bordas baseadas em Geometria Diferencial, a implementação do Sobel em CUDA (CudaSobel) não se fez mais necessária. Entretanto, uma classe ITK foi desenvolvida para essa implementação.

O operador Sobel consiste na aplicação de duas máscaras direcionais sobre a imagem de entrada. Dessa forma obtêm-se as derivadas direcionais horizontal (L_x) e vertical (L_y). A partir dessas derivadas é possível obter a magnitude do gradiente através da Equação 3.5. Enquanto a direção do gradiente (θ) é obtida através da Equação 3.15.

Figura 3.3: MÁSCARAS DO OPERADOR SOBEL.

(a) Sobel X

-1	0	1
-2	0	2
-1	0	1

(b) Sobel Y

1	2	1
0	0	0
-1	-2	-1

Fonte: Autor.

$$\theta = \arctan\left(\frac{L_y}{L_x}\right) \tag{3.15}$$

No *kernel* Sobel, cada pixel é atribuído a uma *thread* que lê o valor de seus vizinhos (vizinhança-8) usando a cache de textura e realiza a convolução de forma semelhante aos *kernels* que calculam as derivadas direcionais no CudaCanny. Com as derivadas direcionais calculadas é possível aplicá-las às equações 3.5 e 3.15 para obter a magnitude e a direção do gradiente.

A classe utilizada como base para a implementação do CudaSobel foi a itkSobelEdgeDetectionImageFilter. A nova classe desenvolvida, itkCudaSobelEdgeDetectionImageFilter, além de executar o cálculo do operador Sobel em CUDA, implementa um método para retornar a direção do gradiente em radianos.

3.3 CLASSE PARA CONFIGURAÇÃO DE KERNEl CUDA NO ITK

Os objetos de imagem no CudaCanny foram gerenciados pela extensão CITK. Como descrito na seção 2.6.2, o CITK modifica a classe *itkImportImageContainer* para suportar *pixel containers* na memória da GPU e manipular as cópias entre memórias, minimizando-as. Entretanto o CITK não possui nenhuma estrutura para armazenar e simplificar configurações de *kernels* CUDA para o ITK. Assim foi desenvolvida uma classe para gerenciar configurações de *kernels* CUDA chamada *itkCudaInterface*. Essa nova classe cria um objeto ITK para armazenar as configurações CUDA. Esta implementação foi criada apenas com os métodos para armazenar as dimensões de blocos e *grids*, mas futuramente configurações mais complexas como o gerenciamento de multi-GPUs, referências de textura, memória compartilhada, *streams* entre outras podem ser implementadas. Assim o gerenciamento das configurações de *kernel* e GPU pode ser unificado em uma estrutura própria do ITK.

3.4 OTIMIZAÇÃO DE ALGORITMOS EM CUDA

A implementação do CudaCanny e do SobelCanny utilizou-se de algumas técnicas para otimizar o desempenho em CUDA. A parte mais importante na otimização de algoritmos em CUDA é o acesso à memória, uma vez que esse tipo de operação costuma ser muito lenta. Outro aspecto relevante é a maximização do paralelismo, ou seja, definir tamanhos certos de blocos e *grids* que mantenham o maior número de *threads* processando e evitar a serialização de *warps*. Técnicas de otimização do uso de instruções que utilizem menos instruções para computar um cálculo são também eficientes. Assim as principais estratégias de otimização

incluem técnicas de:

- Otimização de acessos à memória
- Maximização do paralelismo e evitar a serialização
- Maximização do uso de instruções

Para demonstrar algumas técnicas de otimização baseadas nessas estratégias, foram implementados os seguintes algoritmos que serão descritos nas seções a seguir:

1. Acessos à Memória
2. Serialização de *Warps* e Acessos à Memória
3. Serialização de *Warps* e Expressões Lógicas
4. Configuração de Blocos e *Grid*

3.4.1 Acessos à Memória

O primeiro dos 4 algoritmos foi desenvolvido para demonstrar otimizações no desempenho do acesso à memória. Foram desenvolvidos 3 *kernels* com diversos níveis de otimização para implementar a operação de trocar a ordem dos elementos de um vetor de forma que os elementos das posições pares sejam reposicionados nas posições subsequentes e os elementos das posições ímpares sejam reposicionados nas posições anteriores, como descreve o Sistema de Equações 3.16. Essa operação de troca posição dos elementos de um vetor configura um dos piores casos em relação aos acessos à memória, uma vez que os acessos que trocam a posição dos elementos do vetor ocorrem de forma desalinhada.

$$\begin{cases} out[idx+1] \leftarrow in[idx], & \text{se } idx\%2 = 0 \\ out[idx-1] \leftarrow in[idx], & \text{se } idx\%2 = 1 \end{cases} \tag{3.16}$$

A primeira implementação do *kernel* (1.A) que efetua essa operação é bastante simples. Ela lê o vetor a partir da memória global de forma ordenada e escreve o vetor de saída trocando as posições dos elementos como descrito. A variável **idx**, que indica a posição de cada elemento no vetor, é definida pelas variáveis pré-configuradas de CUDA utilizadas para identificar os blocos (**blockIdx**), as *threads* (**threadIdx**) e o tamanho do bloco (**blockDim**) de forma que cada *thread* acesse uma posição do vetor.

CÓDIGO FONTE 3.1: ALGORITMO 1.A

```
1 __global__ void kernel(float *out, float *in, int N){
2   int idx = blockIdx.x*blockDim.x + threadIdx.x;
3   out[idx-(((idx%2)*2)-1)] = in[idx];
4 }
```

O segundo *kernel* (1.B) que implementa essa operação possui uma diferença sutil para o primeiro. Ao invés de ler o vetor de entrada ordenadamente, essa implementação lê o vetor invertendo as posições para poder escrever os elementos no vetor de saída de forma ordenada.

CÓDIGO FONTE 3.2: ALGORITMO 1.B

```
1 __global__ void fastkernel(float *out, float *in, int N){
2   int idx = blockIdx.x*blockDim.x + threadIdx.x;
3   out[idx] = in[idx-(((idx%2)*2)-1)];
4 }
```

Por fim a terceira implementação desse *kernel* (1.C) utiliza a memória de textura para acessar os elementos do vetor de entrada. Para isso foi utilizada a função **tex1Dfetch** que faz a busca por um elemento em uma referência de textura (**texRef**, neste caso). Essa estratégia foi utilizada em todos os *kernel* implementados no CudaCanny e no SobelCanny.

CÓDIGO FONTE 3.3: ALGORITMO 1.C

```
1 texture<float, 1, cudaReadModeElementType> texRef;
2 __global__ void fastestkernel(float *out, int N){
3   int idx = (blockIdx.x*blockDim.x + threadIdx.x);
4   out[idx] = tex1Dfetch(texRef, idx-(((idx%2)*2-1)));
5 }
```

3.4.2 Serialização de Warps e Acessos à Memória

O segundo algoritmo avalia o acesso à memória e a serialização causada por *threads* divergentes em um mesmo *warp* que ocorre quando *threads* de um mesmo *warp* avaliam de formas diferentes uma expressão condicional (**if/else**). O algoritmo implementado para esse teste utiliza um vetor de entrada para calcular valores para os elementos de outro vetor. O cálculo para cada posição *i* do vetor é definido pela seguinte equação:

$$out[i] = \sqrt{2 * in[i-1] + \frac{in[i+1]}{2}} \tag{3.17}$$

O primeiro e o último elementos, que não possuem um dos elementos vizinhos, utilizam seu próprio valor para o cálculo.

O *kernel* que implementa a primeira versão desse cálculo (2.A) utiliza uma série de operações condicionais para carregar corretamente os valores do vetor de entrada. Com os valores corretos armazenados nos registradores de cada *thread*, o cálculo é realizado e o resultado armazenado no vetor de saída.

CÓDIGO FONTE 3.4: ALGORITMO 2.A

```
__global__ void kernel(float *out, float *in, int N){
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  float2 val;
  if(idx>0) val.x = in[idx-1];
  else val.x = in[idx];
  if(idx<N-1) val.y = in[idx+1];
  else val.y = in[idx];
  if (idx<N) out[idx] = sqrt((2.0*val.x)+(0.5*val.y));
}
```

A segunda implementação desse *kernel* (2.B) realiza os acessos ao vetor através do uso da cache de textura.

CÓDIGO FONTE 3.5: ALGORITMO 2.B

```
texture<float, 1, cudaReadModeElementType> texRef;
__global__ void fastkernel(float *out, int N){
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  float2 val;
  if(idx>0) val.x = tex1Dfetch(texRef, idx-1);
  else val.x = tex1Dfetch(texRef, idx);
  if(idx<N-1) val.y = tex1Dfetch(texRef, idx+1);
  else val.y = tex1Dfetch(texRef, idx);
  if (idx<N) out[idx] = sqrt((2.0*val.x)+(0.5*val.y));
}
```

A terceira implementação deste algoritmo (2.C) utilizou cálculos lógicos para definir a posição a ser acessada pelo *fetch* de textura. Dessa forma não foi necessário armazenar os valores necessários em variáveis, e expressões condicionais que causam serialização de *warps* foram evitadas.

CÓDIGO FONTE 3.6: ALGORITMO 2.C

```
texture<float, 1, cudaReadModeElementType> texRef;
__global__ void fastestkernel(float *out, int N){
```

```
3   int idx = blockIdx.x*blockDim.x + threadIdx.x;
4   if (idx<N){
5     out[idx] = sqrt(2.0*tex1Dfetch(texRef, idx-(idx>0))+(0.5*tex1Dfetch(
          texRef, idx+(idx<N-1))));
6   }
7 }
```

3.4.3 Serialização de Warps e Expressões Lógicas

O terceiro algoritmo avalia a serialização das *threads* de cada *warp*. Para isso, cada *thread* calcula o valor da variável **theta** utilizando o elemento da posição correspondente ao índice **idx** da *thread* no vetor de entrada **in[]** conforme a Equação 3.18. E, de acordo com a Equação 3.19, atribui valores às variáveis **dir.x** e **dir.y**. Ao final as duas variáveis são somadas e o valor resultante atribui valor a uma posição do vetor de saída.

$$theta = \left\lceil \frac{(in[idx] - 22)}{45} \right\rceil \tag{3.18}$$

$$\begin{cases} dir.x \leftarrow 0 \text{ e } dir.y \leftarrow -1, & \text{se } (threadIdx.x\%2 = 0)\&(theta = 0) \\ dir.x \leftarrow -1 \text{ e } dir.y \leftarrow -1, & \text{se } (threadIdx.x\%2 = 0)\&(theta = 1) \\ dir.x \leftarrow 1 \text{ e } dir.y \leftarrow 0, & \text{se } (threadIdx.x\%2 = 0)\&(theta = 2) \\ dir.x \leftarrow 0 \text{ e } dir.y \leftarrow -1, & \text{se } (threadIdx.x\%2 = 0)\&(theta = 3) \\ dir.x \leftarrow 1 \text{ e } dir.y \leftarrow 0, & \text{se } (threadIdx.x\%2 \neq 0)\&(theta = 0) \\ dir.x \leftarrow 0 \text{ e } dir.y \leftarrow 0, & \text{se } (threadIdx.x\%2 \neq 0)\&(theta = 1) \\ dir.x \leftarrow 2 \text{ e } dir.y \leftarrow 1, & \text{se } (threadIdx.x\%2 \neq 0)\&(theta = 2) \\ dir.x \leftarrow 2 \text{ e } dir.y \leftarrow 0, & \text{se } (threadIdx.x\%2 \neq 0)\&(theta = 3) \end{cases} \tag{3.19}$$

Inicialmente o *kernel* (3.A) foi implementado usando expressões condicionais para definir os valores das duas variáveis que são somadas ao final da execução.

CÓDIGO FONTE 3.7: ALGORITMO 3.A

```
1 __global__ void kernel(float *out, float *in, int N){
2   int idx = blockIdx.x*blockDim.x + threadIdx.x;
3   float2 dir;
4   float theta = ceilf((in[idx]-22)/45);
5   if (threadIdx.x%2){
```

```
6       if (theta == 0){
7         dir.x = 0;
8         dir.y = -1;
9       }
10      if (theta == 1){
11        dir.x = dir.y = -1;
12      }
13      if (theta == 2){
14        dir.x = 1;
15        dir.y = 0;
16      }
17      if (theta == 3){
18        dir.x = 1;
19        dir.y = -1;
20      }
21    }
22    else{
23      if (theta == 0){
24        dir.x = 1;
25        dir.y = 0;
26      }
27      if (theta == 1){
28        dir.x = dir.y = 0;
29      }
30      if (theta == 2){
31        dir.x = 2;
32        dir.y = 1;
33      }
34      if (theta == 3){
35        dir.x = 2;
36        dir.y = 0;
37      }
38    }
39    out[idx] = dir.x+dir.y;
40  }
```

Outra implementação do *kernel* (3.B) utilizou expressões lógicas para calcular os valores das variáveis que serão somadas, definindo os elementos do vetor de saída da seguinte forma:

CÓDIGO FONTE 3.8: ALGORITMO 3.B

```
1 __global__ void fastestkernel(float *out, float *in, int N){
```

```
2   int idx = blockIdx.x*blockDim.x + threadIdx.x;
3   float theta = ceilf(__fdividef(in[idx]-22,45));
4   out[idx] = ((1-(!theta)-((theta == 1)<<1))+(theta == 2)-1)+(!(threadIdx.x
        %2))<<1;
5 }
```

3.4.4 Configuração de Blocos e Grid

O último dos 4 algoritmos mostra a importância da configuração dos blocos e *grid* ao invocar um *kernel* para a maximização do paralelismo em CUDA. Nesse teste apenas um *kernel* foi desenvolvido e utilizado para avaliar duas configurações de blocos e *grid*. O algoritmo utilizado simplesmente realiza a cópia de cada elemento de um vetor para outro vetor incrementando os valores de cada elemento em 1. O código implementado para esse teste foi o seguinte:

CÓDIGO FONTE 3.9: ALGORITMO 4

```
1 __global__ void kernel(float *out, float *in, int N){
2   int idx = blockIdx.x*blockDim.x + threadIdx.x;
3   if (idx<N) out[idx] = in[idx]+1.f;
4 }
```

4 MATERIAIS E MÉTODOS

Neste capítulo serão descritos as imagens e equipamentos utilizados e os testes realizados. Os testes foram divididos em três conjuntos de testes. Testes de Qualidade, Testes de Desempenho e Testes de Algoritmo. Os Testes de Qualidade foram utilizados para medir a qualidade das bordas detectadas pelo CudaCanny em relação ao Canny do ITK (ItkCanny) a fim de verificar a possibilidade de substituir um pelo outro. Os Testes de Desempenho compararam a velocidade de execução dos dos algoritmos para um conjunto de dados em diferentes máquinas. Embora o modelo de programação em placas gráficas e o tradicional sejam diferentes e a sua comparação não-trivial, a avaliação de desempenho proposta se faz útil na medida que a implementação do CudaCanny visa fornecer um meio de obter os mesmos resultados do ItkCanny em um tempo de execução menor. E os Testes de Algoritmo avaliaram diferentes técnicas de otimização de código em CUDA. Para os Testes de Qualidade e Desempenho, foram utilizados os seguintes parâmetros no algoritmo de Canny definidos empiricamente com o objetivo de obter um nível de detecção adequado a todas as imagens testadas:

- σ = 1,4
- T_h = 7
- T_l = 4

As implementações do CudaCanny e do CudaSobel utilizaram como estratégia o processamento de um pixel por processador CUDA. Essa estratégia é vista como ideal por Bräunl (2001). Assim cada *thread* é relacionada a um pixel de forma que todos os pixels da imagem sejam processados. As imagens tanto no ITK quanto nas implementações deste trabalho são definidas como vetores unidimensionais de pixels. Portanto os *kernels* de processamento de imagens utilizaram configurações de *grid* e blocos unidimensionais de forma semelhante ao conjunto de dados processado. Os blocos foram configurados com 256 *threads*, enquanto que a quantidade de blocos por *grid* utilizada é definida pela Equação 4.1, onde N_{Bl} representa a

quantidade de blocos necessária para processar N_{Px} pixels utilizando a configuração de N_{Th} *threads* por bloco. Essa configuração foi definida de forma a maximizar a ocupação da GPU e otimizar a execução.

$$N_{Bl} = \frac{N_{Px} + N_{Th} - 1}{N_{Th}} \tag{4.1}$$

4.1 BASES DE IMAGENS

As bases de imagens usadas para os Testes de Qualidade e Desempenho são:

Tabela 4.1: BASES DE IMAGENS.

BASE	RESOLUÇÃO DAS IMAGENS EM PIXELs	QUANTIDADE DE IMAGENS
B1	321×481 e 481×321	100
B2	642×962 e 962×642	100
B3	1284×1924 e 1924×1284	100
B4	2568×3848 e 3848×2568	100

A base B1 foi criada a partir das imagens de teste da base de dados *Berkeley Segmentation Dataset* (MARTIN et al., 2001) que encontra-se disponível (gratuitamente para fins não-lucrativos e para fins educacionais) em (ARBELAEZ; FOWLKES; MARTIN, 2007). Para gerar a base B1 as 100 imagens de teste da *Berkeley Segmentation Dataset* foram convertidos de JPG para PNG em tons de cinza.

Figura 4.1: EXEMPLO DE REPLICAÇÃO UTILIZADA PARA CRIAR IMAGENS MAIORES.

(a) Original (b) Replicada

Fonte: Autor.

A partir da base B1, foram criadas bases com imagens maiores para avaliar o processa-

mento de grandes quantidades de dados nos Testes de Desempenho. Assim uma nova base foi criada a partir da replicação das imagens da B1. Cada imagem da base B2 consiste na replicação horizontal e vertical de cada imagem da base B1, como mostra a Figura 4.1. Assim, as imagens da base B2 possui quatro vezes mais pixels que as imagens da base B1. Esse processo foi repetido na base B2 para criar a base B3. E na base B3 para criar a base B4. Todas as operações de replicação e conversão das imagens foram realizadas através do comando *convert* contido na suíte de manipulação de imagens *Image Magick* (STILL, 2005). O seguinte comando foi utilizado para criar a base B1:

CÓDIGO FONTE 4.1: SCRIPT PARA GERAÇÃO DA BASE B1

```
1 #!/bin/bash
2
3 for i in *.jpg; do
4   convert $i -colorspace gray B1/${i/.jpg/.png};
5 done
```

E o seguinte comando foi utilizado para criar as demais bases:

CÓDIGO FONTE 4.2: SCRIPT PARA A GERAÇÃO DAS BASES B2 B3 E B4

```
1 #!/bin/bash
2
3 [ "$1" ] && SRC=$1 || SRC='./'
4 [ "$2" ] && DEST=$2 || DEST='./'
5
6 for i in $SRC/*.png; do
7   convert $i $i -append ${i/$SRC/tt};
8   convert ${i/$SRC/tt} ${i/$SRC/tt} +append ${i/$SRC/$DEST};
9 done
```

4.2 HARDWARE

Para os Testes de Desempenho e de Algoritmo foram utilizadas dois computadores: um servidor e um *desktop* com as seguintes configurações de *hardware*:

Servidor:

- CPU: 2x Intel®Core™i7 3,3GHz com 4 núcleos cada, 8192KB de cache e 12GB de RAM
- GPU1: NVidia Tesla C2050 com 448 núcleos de 1,15GHz e 3GB de RAM.

- GPU2: NVidia Tesla C1060 com 240 núcleos de 1,3GHz e 4GB de RAM.

Desktop:

- CPU: Intel®Core™2Duo E7400 2,80GHz com 3072KB de cache e 2GB de RAM
- GPU: NVidia GeForce 8800 GT com 112 núcleos de 1,5GHz e 512MB de RAM.

Apenas o *desktop* foi utilizado para o teste de Qualidade, uma vez que a diferença do *hardware* não influencia o resultado desse teste. Os equipamentos usados nos testes pertencem aos grupos C3SL e VRI do Departamento de Informática da Universidade Federal do Paraná. Elas não possuem disco local. Os dados são armazenados em uma servidora de dados e transferidos por uma rede *Gigabit*. O tempo de carga das imagens para o disco não é computado.

4.3 TESTES DE QUALIDADE

Os Testes de Qualidade visam avaliar a relação de qualidade entre as bordas detectadas pelo ItkCanny e CudaCanny. A qualidade das bordas detectadas é avaliada através da comparação das bordas detectadas por um dado detector com uma imagem ideal de referência. Existem vários critérios para avaliar um detector de bordas (BOAVENTURA; GONZAGA, 2009). Podem ser utilizadas medidas diretas como a quantidade de pixels de bordas corretamente detectadas (TP), a quantidade de pixels de bordas não detectadas (FN), e a quantidade de pixels de bordas erroneamente detectadas (FP). Através dessas medidas chegamos às métricas encontradas em (BOAVENTURA; GONZAGA, 2009) que foram utilizadas neste teste:

A Porcentagem de pixels de borda detectados corretamente (P_{co}):

$$P_{co} = \frac{TP}{max(N_I, N_B)} \tag{4.2}$$

Onde N_I representa a quantidade de bordas de referência e N_B a quantidade de bordas detectadas.

A Porcentagem de pixels de borda não detectados (P_{nd}):

$$P_{nd} = \frac{FN}{max(N_I, N_B)} \tag{4.3}$$

E a Porcentagem de falsos alarmes (P_{fa}), ou seja, pixels que foram detectados, mas que não fazem parte do conjunto dos pixels de borda de referência:

$$P_{fa} = \frac{FP}{max(N_I, N_B)} \tag{4.4}$$

Uma vez que um dos objetivos deste trabalho é implementar uma versão em Cuda do detector de bordas Canny que seja equivalente à implementação encontrada na biblioteca ITK, as bordas detectadas pelo ItkCanny foram consideradas bordas de referência. Isso significa que quanto maior o valor do P_{co} e menor os valores de P_{nd} e P_{fa} mais semelhantes as bordas detectadas pelo CudaCanny são das bordas detectadas pelo ItkCanny. De forma que se os valores forem satisfatórios pode-se concluir que os detectores sejam equivalentes e portanto que a implementação do CudaCanny é válida como um substituto do ItkCanny.

O cálculo dessas métricas é feito para cada imagem de cada base. A média das métricas das imagens de uma base inteira é usada para medir a qualidade das bordas detectadas nesta base base.

4.4 TESTES DE DESEMPENHO

Os Testes de Desempenho tem por objetivo avaliar o tempo de execução dos detectores de borda CudaCanny e ItkCanny. Foram medidos os tempos de execução total e parcial de cada detector. O tempo total corresponde à soma dos tempos de execução de um detector para todas as imagens de uma base. Cada medida de tempo parcial corresponde à soma dos tempos de execução de uma etapa do detector em todas as imagens de uma base. As bases utilizadas para esses testes foram as descritas na seção 4.1. Os tempos total e parcial da execução de cada detector foram medidos 100 vezes para cada base.

Os tempos na implementação em CUDA foram medidos utilizando funções da biblioteca *cutil* que está incluída no *kit* de desenvolvimento de CUDA. Foram incluídas medições de tempo também na implementação do ITK, utilizando funções da biblioteca *time.h*. Em ambas bibliotecas a medida utilizada é a de tempo decorrido. Os tempos foram medidos em ambas implementações nos mesmos pontos do código.

O tempo de execução total é calculado através da medida do tempo da execução do método *Update()*. Para o CudaCanny esse tempo é somado à medida do tempo de cópia dos dados da memória da GPU para a memória da CPU. O tempo dessa cópia é somado ao tempo de execução do método *Update()* pois ela não ocorre durante o método como a cópia dos dados da memória da CPU para a memória da GPU que ocorre na Gaussiana. Dessa forma o tempo total exclui o tempo de carregamento do arquivo de imagem para a memória, feito em uma operação anterior, e a escrita do resultado em outro arquivo de imagem. O código exibido a seguir mostra como é realizada a medida dos tempos de execução nos detectores CudaCanny e ItkCanny para os Testes de Desempenho:

CÓDIGO FONTE 4.3: MEDIÇÃO DO TEMPO TOTAL

```
1   unsigned int CannyTimer = 0;
2   cutCreateTimer( &CannyTimer ); // CRIA TIMER
3
4   CannyFilter::Pointer canny = CannyFilter::New();
5   canny->SetInput(reader->GetOutput());
6   canny->SetVariance(gaussianVariance);
7   canny->SetUpperThreshold(t2);
8   canny->SetLowerThreshold(t1);
9
10  cutStartTimer( CannyTimer ); // INICIA TIMER
11
12  canny->Update();
13
14  cutStopTimer( CannyTimer ); // PARA TIMER
15  printf("cudaCanny time: %f ms\n",cutGetTimerValue( CannyTimer ));
16 }
```

Os tempos de cópia dos dados entre as memórias são medidos direto em seus respectivos métodos, como mostra o código a seguir:

CÓDIGO FONTE 4.4: MEDIÇÃO DOS TEMPOS DE CÓPIA DE DADOS ENTRE MEMÓRIAS

```
1  template <typename TElementIdentifier, typename TElement>
2  void CudaImportImageContainer< TElementIdentifier , TElement >
3  ::CopyToGPU() const{
4    std::cout << serial << " Copying to GPU " << std::endl;
5    AllocateGPU();
6
7    unsigned int timer = 0;
8    cutCreateTimer( &timer ); // CRIA TIMER
9    cutStartTimer( timer );  // INICIA TIMER
10
11   cudaMemcpy(m_DevicePointer, m_ImportPointer,
12               sizeof(TElement)*m_Size, cudaMemcpyHostToDevice);
13
14   cutStopTimer( timer );  // PARA TIMER
15   printf("Input time = %f ms\n",cutGetTimerValue( timer ));
16
17   ImageLocation = BOTH;
18 }
19
```

```
20 template <typename TElementIdentifier, typename TElement>
21 void CudaImportImageContainer< TElementIdentifier , TElement >
22 ::CopyToCPU() const{
23   std::cout << serial << " Copying to CPU " << std::endl;
24
25   unsigned int timer = 0;
26   cutCreateTimer( &timer ); // CRIA TIMER
27   cutStartTimer( timer );  // INICIA TIMER
28
29   cudaMemcpy(m_ImportPointer, m_DevicePointer,
30              sizeof(TElement)*m_Size, cudaMemcpyDeviceToHost);
31
32   cutStopTimer( timer );  // PARA TIMER
33   printf("Output time = %f ms\n",cutGetTimerValue( timer ));
34
35   ImageLocation = BOTH;
36 }
```

Os tempos parciais foram medidos no método *GenerateData()* de ambas implementações do Canny. Da mesma forma que na medida do tempo total, os tempos parciais começaram a ser medidos imediatamente antes de cada método *Update()*, no caso da Gaussiana e do *Zero Crossing*, e antes das chamadas dos métodos das derivadas e da histerese. Com isso o tempo de execução da Gaussiana no CudaCanny inclui a cópia da imagem para a memória da GPU. Assim, o tempo da cópia é subtraído do tempo de execução da Gaussiana no CudaCanny.

A medição dos tempos parciais é interrompida logo em após a execução dos métodos. Dessa forma a diferença entre o tempo total e a soma dos tempos parciais é devida ao gerenciamento de fluxos das classes CudaCanny e ItkCanny, que não é considerado. No entanto o gerenciamento de fluxo do ITK é considerado na medida dos tempos para as classes da Gaussiana e do *Zero Crossing*. O código a seguir mostra como foram inseridas essas medições no filtro do CudaCanny, elas foram inseridas em posições equivalentes no filtro do ItkCanny:

CÓDIGO FONTE 4.5: MEDIÇÃO DOS TEMPOS PARCIAIS

```
1   m_CudaGaussianFilter->SetInput(input);
2
3   unsigned int timer = 0;
4   cutCreateTimer( &timer ); // CRIA TIMER
5   cutStartTimer( timer ); // INICIA TIMER
6
7   m_CudaGaussianFilter->Update();
8
```

```
9    cutStopTimer( timer ); // PARA TIMER
10   printf("Gaussian time = %f ms\n",cutGetTimerValue( timer ));
11
12   timer = 0;
13   cutCreateTimer( &timer ); // CRIA TIMER
14   cutStartTimer( timer ); // INICIA TIMER
15
16   this->Cuda2ndDerivative();
17
18   cutStopTimer( timer ); // PARA TIMER
19   printf("2nd Derivative time = %f ms\n",cutGetTimerValue( timer ));
20
21   m_CudaZeroCrossingFilter->SetInput(this->GetOutput());
22
23   timer = 0;
24   cutCreateTimer( &timer ); // CRIA TIMER
25   cutStartTimer( timer ); // INICIA TIMER
26
27   m_CudaZeroCrossingFilter->Update();
28
29   cutStopTimer( timer ); // PARA TIMER
30   printf("Maximum Supression time = %f ms\n",cutGetTimerValue( timer ));
31
32   timer = 0;
33   cutCreateTimer( &timer ); // CRIA TIMER
34   cutStartTimer( timer ); // INICIA TIMER
35
36   this->CudaHysteresisThresholding();
37
38   cutStopTimer( timer ); // PARA TIMER
39   printf("Hysteresis time = %f ms\n",cutGetTimerValue( timer ));
40
41 }
```

Para garantir que nenhum tipo de *overhead* de inicialização das memória das placas gráficas interfira na medição dos tempo, uma operação de ”aquecimento” (*warm up*) é realizada antes de cada execução no CudaCanny (PODLOZHNYUK, 2007). Dessa forma esse tempo é desconsiderado da medida do tempo de execução do CudaCanny. A operação de aquecimento consiste em alocar e liberar uma região de memória da placa gráfica da seguinte forma:

CÓDIGO FONTE 4.6: OPERAÇÃO DE ”AQUECIMENTO”DA GPU

```
1    int *rub;
```

```
2  cudaMalloc( (void**)&rub, reader->GetOutput()->GetPixelContainer()->Size
       () * sizeof(int) );
3  cudaFree( rub );
```

São comparados o desempenho entre as execuções em cada uma das bases, a quantidade de pixels processados por tempo para cada execução e a proporção dos tempos parciais de cada etapa do Canny. A comparação de desempenho é calculada pela divisão entre os tempos totais de execução de uma base. Enquanto a quantidade de pixels por tempo é calculada dividindo a soma da quantidade de pixels das imagens de uma base pelo seu tempo total de execução em milissegundos. Com a proporção dos tempos parciais de cada etapa do Canny é possível mostrar o quanto as etapas ocupam de processamento para cada execução.

4.5 TESTES DE ALGORITMO

Os Testes de Algoritmo objetivam demonstrar e avaliar técnicas de otimização para CUDA através da comparação do tempo de execução de certos trechos de código. Para isso, cada versão dos algoritmos descritos na seção 3.4 foi executada 100 vezes, o tempo de execução de cada *kernel* foi medido e a sua média utilizada para avaliar a eficiência de cada estratégia aplicada. Para garantir que nenhum *overhead* de inicialização das memória interfira nas medições de tempo, a operação de ”aquecimento”, descrita na seção anterior também foi aplicada antes de cada execução. Os resultados desse teste foram avaliados através dos tempos de execução de cada otimização e também para as diferentes arquiteturas de placa gráfica.

4.5.1 Teste de Acessos à Memória

Este teste avalia o desempenho das técnicas de acesso à memória através dos *kernels* que implementam a troca de posição dos elementos de um vetor descritos na seção 3.4.1. O vetor utilizado possui 10 milhões de posições em **float** (32 bits) contendo valores inteiros de 0 a 999 gerados aleatoriamente[1]. Em todas as implementações os *kernels* foram invocados com a configuração de 256 *threads* por bloco e a quantidade de blocos foi definida pela Equação 4.1 (ver página 40).

[1]Para gerar os valores aleatoriamente foi utilizada a função rand() da biblioteca libc.

4.5.2 Teste de Serialização de Warps e Acessos à Memória

Para este teste foram utilizadas as versões do algoritmo descrito na seção 3.4.2 que avaliam técnicas para resolver os problemas de serialização de *warps* e de otimização dos acessos à memória. O algoritmo utiliza um vetor de entrada para definir os valores utilizados no cálculo da Equação 3.17 (ver página 34). O vetor utilizado neste teste possui 10 milhões de posições em **float** (32 bits) cujos elementos são calculados aleatoriamente contendo valores inteiros de 0 a 999. Em todas as implementações deste algoritmo os *kernels* foram invocados com a configuração de 256 *threads* por bloco e a quantidade de blocos calculada pela Equação 4.1.

4.5.3 Teste de Serialização de Warps e Expressões Lógicas

As implementações do algoritmo deste teste, descritas na seção 3.4.3, avaliam o uso de expressões lógicas para resolver o problema de serialização de *warps*. Os valores de um vetor de entrada são utilizados para realizar o cálculo da Equação 3.18. O vetor de entrada possui 10 milhões de elementos em **float** (32 bits) com valores variando de 0 a 157. Ambas as implementações deste algoritmo os *kernels* foram invocados com a configuração de 256 *threads* por bloco e quantidade de blocos calculada pela Equação 4.1.

4.5.4 Teste de Configuração de Blocos e Grid

Para este teste foi utilizado o algoritmo definido na seção 3.4.4 que realiza a cópia de elementos entre dois vetores incrementando o valor de cada elemento em 1. O teste avalia duas configurações de blocos e *grid*. Um vetor de entrada com 100000 elementos em **float** (32 bits) com valores variando entre 0 e 179. A configuração da *grid* consiste na quantidade de *threads* por blocos e de blocos. A quantidade de blocos é definida pela Equação 4.1. Dessa forma, no primeiro momento é realizado o teste (4.A) de uma *grid* com 5 *threads* por bloco e 20000 blocos. E em seguida outro teste (4.B) com 256 *threads* por bloco e 391 blocos.

5 RESULTADOS EXPERIMENTAIS

Neste capítulo encontram-se os resultados dos testes descritos no capítulo 4.

5.1 TESTES DE QUALIDADE

A execução dos Testes de Qualidade realizado entre os detectores CudaCanny e ItkCanny mostraram os seguintes resultados:

Tabela 5.1: RESULTADOS DOS TESTES DE QUALIDADE.

BASE	P_{co}	P_{nd}	P_{fa}
B1	0,9947	0,0043	0,0050
B2	0,9970	0,0027	0,0022
B3	0,9981	0,0018	0,0011
B4	0,9989	0,0010	0,0005

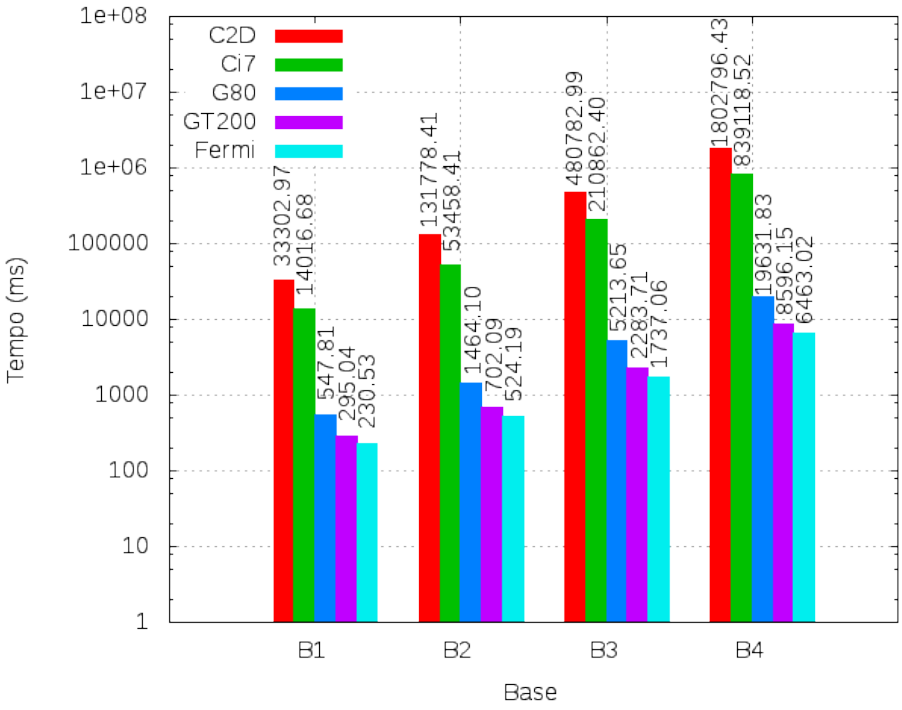
A primeira coluna da tabela identifica quais as bases foram testadas. As demais colunas mostram os resultados dos testes para cada métrica. Os valores contidos nelas variam de 0 a 1. Na segunda coluna encontram-se os resultados de bordas que foram identificadas em ambos detectores testados (P_{co}). A terceira coluna mostra os resultados para as bordas que foram detectadas apenas na imagem de referência (P_{nd}), ou seja, as bordas detectadas pelo ItkCanny. A quarta e última coluna mostra os resultados para os falsos alarmes (Pfa), ou seja, as bordas que foram detectadas apenas pelo detector que está sendo avaliado, mas não existem na imagem de referência.

5.2 TESTES DE DESEMPENHO

O gráfico da Figura 5.1 compara os tempos de execução entre o CudaCanny e o ItkCanny em cada um dos computadores utilizados para cada uma das bases de imagens. As barras vermelhas representam o tempo de execução do Canny do ITK na CPU Core ^{TM}2 Duo (C2D) presente no *desktop*. As barras verdes representam o tempo de execução do ItkCanny nas CPUs Core ^{TM}i7 (Ci7) do servidor. As barras azul escuro, violeta e azul claro representam dos tempo

de execução do cudaCanny respectivamente nas placas gráficas GeForce 8800 GT (G80) do *desktop*, Tesla C1060 (GT200) e Tesla C2050 (Fermi) do servidor. A Tabela 5.2 mostra os tempos de execução das 100 imagens de cada base em milissegundos e o desvio padrão.

Figura 5.1: TEMPO DE EXECUÇÃO.



Fonte: Autor.

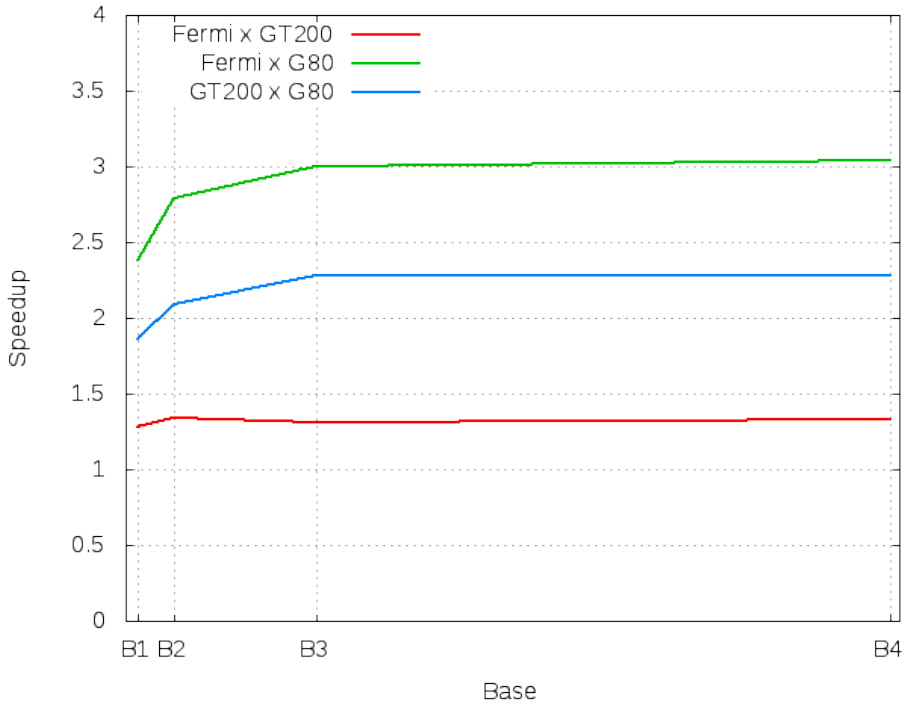
Tabela 5.2: RESULTADOS DOS TESTES DE DESEMPENHO.

HARDWARE	BASE	TEMPO DE EXECUÇÃO (ms)	DESVIO PADRÃO
C2D	B1	33302,97	534,30 (1,60%)
	B2	131778,41	2085,26 (1,58%)
	B3	480782,99	3281,00 (0,68%)
	B4	1802796,43	3913,80 (0,21%)
Ci7	B1	14016,68	46,89 (0,33%)
	B2	53458,41	52,31 (0,09%)
	B3	210862,40	130,65 (0,06%)
	B4	829118,52	495,12 (0,5%)
G80	B1	547,81	1,66 (0,30%)
	B2	1464,10	1,48 (0,10%)
	B3	5213,65	27,90 (0,53%)
	B4	19631,83	13,32 (0,06%)
GT200	B1	295,04	0,46 (0,15%)
	B2	702,09	1,08 (0,15%)
	B3	2283,71	4,22 (0,18%)
	B4	8596,15	15,41 (0,17%)
Fermi	B1	230,53	0,66 (0,29%)
	B2	524,19	0,73 (0,13%)
	B3	1737,06	90,68 (5,22%)
	B4	6463,02	248,05 (3,83%)

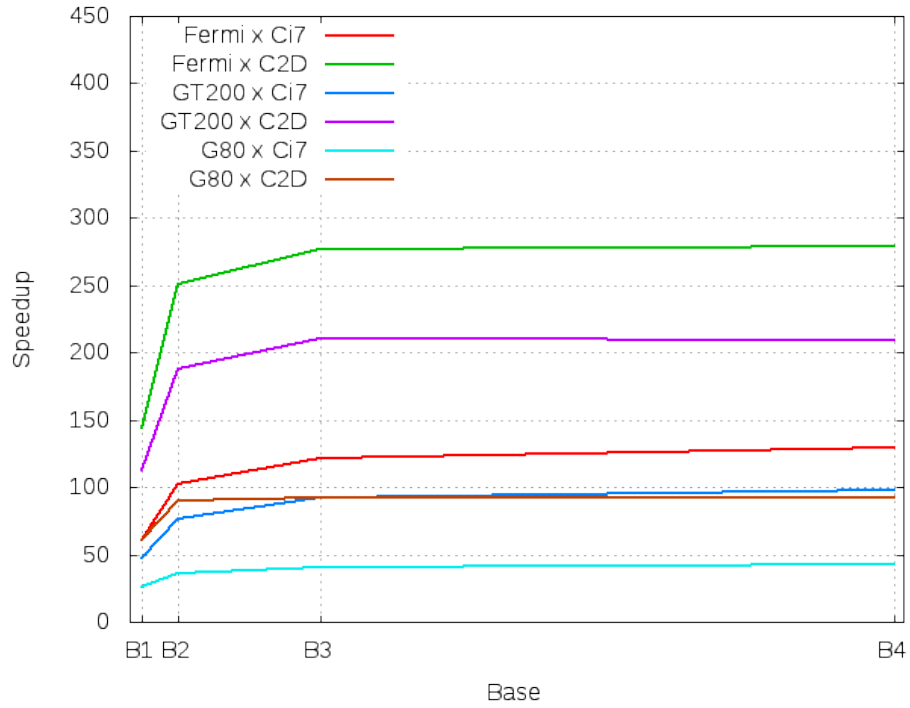
O Gráfico da Figura 5.2 mostra o ganho de desempenho do CudaCanny entre as diferentes placas gráficas utilizadas e com as execuções do ItkCanny em cada uma das bases. A distância entre as bases no eixo horizontal mostra a diferença entre a soma da quantidade de pixels das imagens de cada base.

Figura 5.2: GANHO DE DESEMPENHO (*SPEEDUP*) ENTRE AS EXECUÇÕES.

(a) GPUs × GPUs



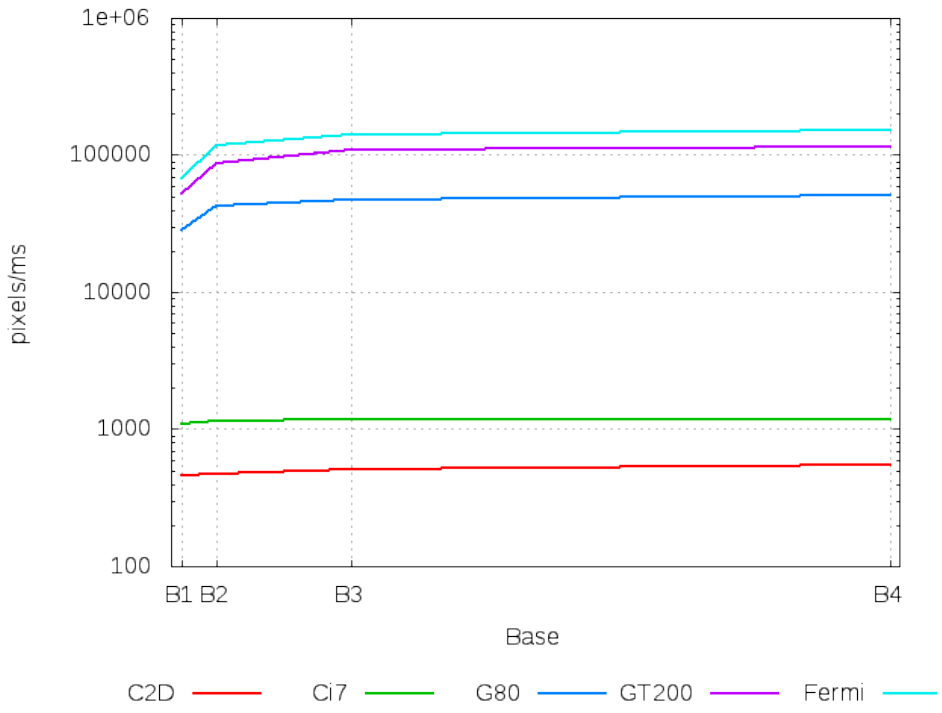
(b) GPUs × CPUs



Fonte: Autor.

O Gráfico da Figura 5.3 mostra a quantidade de pixels processados pelas execuções em cada uma das bases.

Figura 5.3: QUANTIDADE DE PIXELS PROCESSADOS POR MILISSEGUNDO.

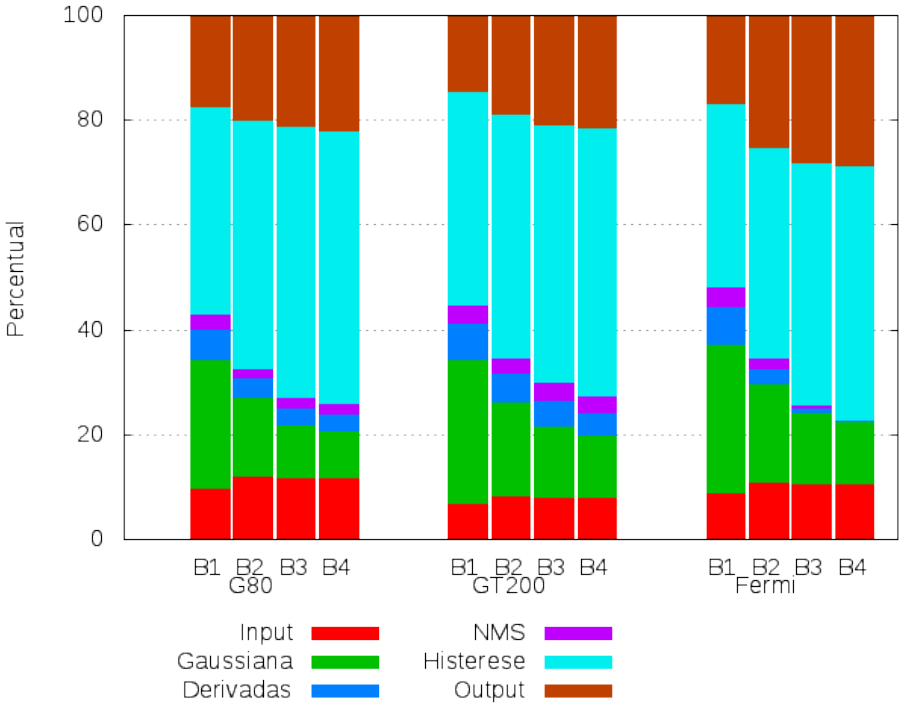


Fonte: Autor.

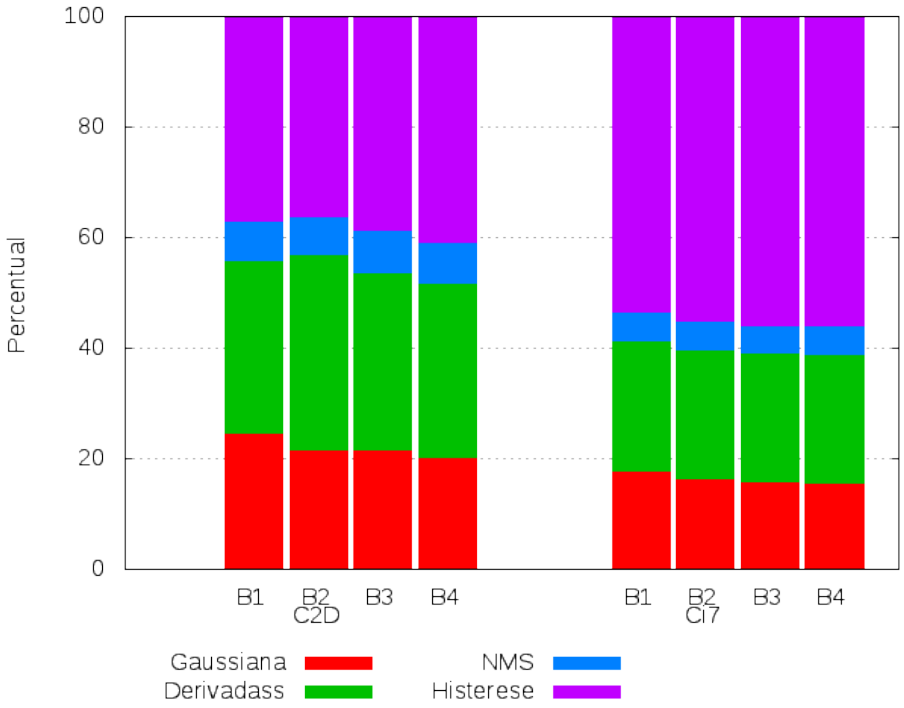
O Gráfico da Figura 5.4 mostra a porcentagem gasta em cada etapa do Canny nas implementações em cada computador ao processar cada uma das bases.

Figura 5.4: PERCENTUAL DAS PARCIAIS DAS EXECUÇÕES.

(a) GPUs



(b) CPUs



Fonte: Autor.

5.3 TESTES DE ALGORITMO

A Tabela 5.3 com os resultados dos Testes de Algoritmo mostra, para cada um dos algoritmos, uma breve descrição e os tempos de execução para cada otimização e para cada uma das arquiteturas.

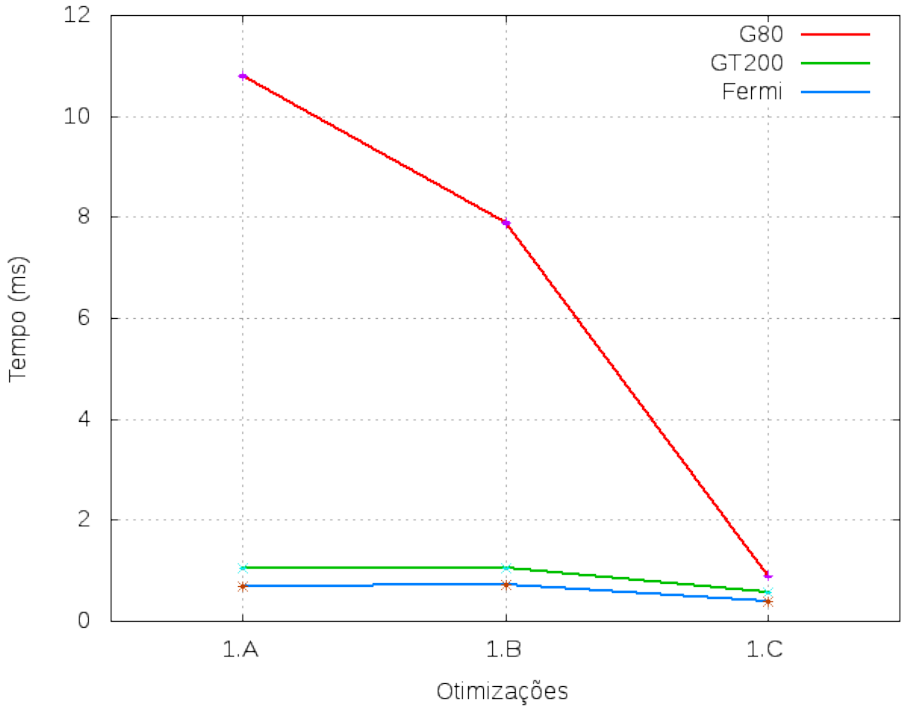
Os Testes de Algoritmo mostraram os seguintes resultados:

Tabela 5.3: RESULTADOS DOS TESTES DE ALGORITMO.

ALG. (SEÇÃO)	DESCRIÇÃO/IMPLEMENTAÇÕES	TEMPO EM ms (DESVIO PADRÃO)		
		G80	GT200	FERMI
3.4.1	1.A) Troca elementos de um vetor a cada 2 posições.	10,79 (0,2%)	1,04 (0,1%)	0,68 (0,1%)
	1.B) Troca posições durante a leitura e não na escrita.	7,88 (0,4%)	1,04 (0,1)	0,71 (0,1%)
	1.C) Uso de cache de Textura.	0,88 (0,7%)	0,55 (0,2%)	0,39 (0,3%)
3.4.2	2.A) Calcula $\sqrt{2 * in[i-1] + \frac{in[i+1]}{2}}$	14,86 (0,2%)	2,2 (0,1%)	0,90 (0,2%)
	2.B) Uso de cache de Textura.	2,21 (0,3%)	1,16 (0,7%)	0,92 (0,3%)
	2.C) Calcula posições com expressões lógicas.	1,69 (0,4%)	1,06 (0,09%)	0,82 (0,2%)
3.4.3	3.A) Calcula dir.x+dir.y	2,56 (0,1%)	1,31 (0,9%)	1,27 (0,08%)
	3.B) Calcula valores com expressões lógicas.	1,63 (0,2%)	1,04 (0,1%)	0,76 (0,2%)
3.4.4	$out[idx] \leftarrow in[idx] + 1$			
	4.A) $N_{Th} = 5$, $N_{Bl} = 20000$	0,17 (0,9%)	0,15 (0,4%)	0,15 (0,3%)
	4.B) $N_{Th} = 256$, $N_{Bl} = 391$	0,03 (3,0%)	0,025 (2,3%)	0,016 (4,7%)

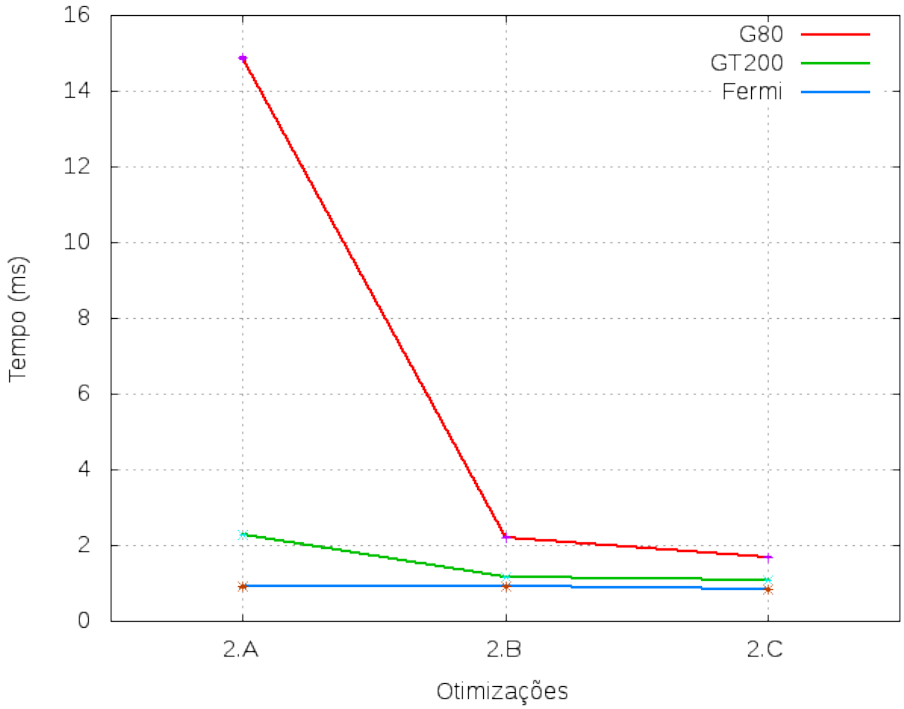
Os Gráficos das Figuras 5.5, 5.6, 5.7 e 5.8 mostram os tempos de execução apresentados pelas implementações em cada uma das placas gráficas para cada um dos respectivos algoritmos.

Figura 5.5: RESULTADO DOS TESTES DE ALGORITMO - ACESSOS À MEMÓRIA.



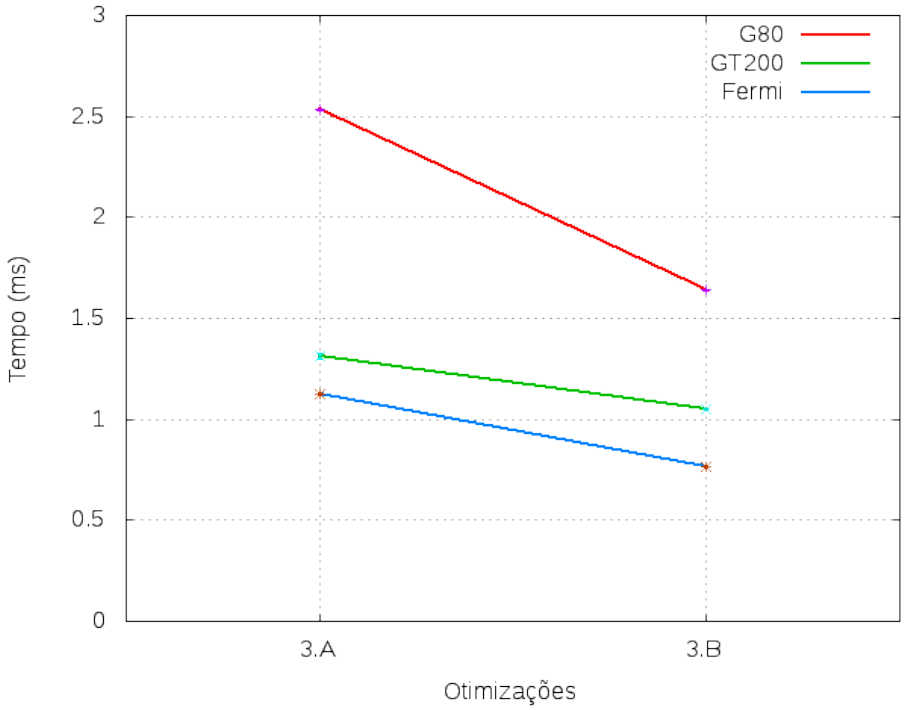
Fonte: Autor.

Figura 5.6: RESULTADO DOS TESTES DE ALGORITMO - SERIALIZAÇÃO DE WARPS E ACESSOS À MEMÓRIA.



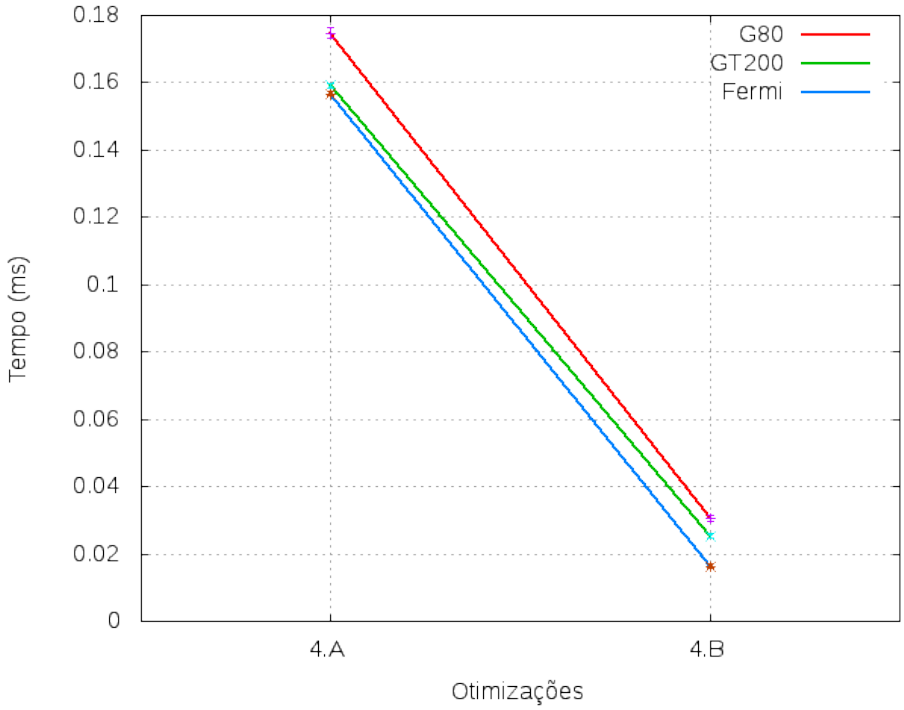
Fonte: Autor.

Figura 5.7: RESULTADO DOS TESTES DE ALGORITMO - SERIALIZAÇÃO DE WARPS E EXPRESSÕES LÓGICAS.



Fonte: Autor.

Figura 5.8: CONFIGURAÇÃO DE BLOCOS E GRID.



Fonte: Autor.

6 DISCUSSÃO

Neste capítulo serão discutidos os aspectos pertinentes à criação das bases de imagens e aos resultados obtidos nos testes.

6.1 BASES DE IMAGENS

As diferentes bases de imagens foram criadas a partir da base de imagens de teste da *Berkeley Segmentation Dataset* para que os Testes de Desempenho pudessem avaliar o desempenho dos algoritmos com diferentes quantidades de dados. A base contendo as maiores imagens foi a B4. A criação de uma base B5 seguindo os mesmos procedimentos descritos na seção 4.1 usados para criar as bases B2, B3 e B4 não foi possível devido à abordagem de implementação do CudaCanny, no qual cada pixel é processado individualmente por uma *thread* em blocos de 256 *threads*. Cada imagem da B4 possui 9881664 pixels e necessita de 38601 blocos para ser processada. Sabendo que as GPUs de todas as arquiteturas da NVidia possuem a limitação física de 65536 blocos e que uma imagem da base B5 conteria 39526656 pixels (4 vezes o tamanho de uma imagem da B4), seriam necessitários 154401 blocos para processar uma imagem dessa base. Portanto, devido à essa limitação, é possível processar imagens com no máximo 16777216 pixels, o que equivale a uma imagem de 4096×4096 pixels de resolução. Para imagens maiores seria necessário reimplementar o CudaCanny de forma a dividir as imagens em fatias de até 16777216 pixels e processá-las sequencialmente[1].

6.2 TESTES DE QUALIDADE

Os resultados exibidos na Tabela 5.1 mostram que para todas as bases a porcentagem de pixels de borda detectados (P_{co}) em ambas imagens foi superior a 99%. Enquanto que os dados das métricas P_{nd} e P_{fa} mostram que menos de 1% dos pixels de bordas detectados em apenas uma das imagens. Esses resultados mostram que quase a totalidade das bordas identificadas

[1] A extensão CITK possui um método que permite a divisão de uma imagem em fatias para serem processadas sequencialmente.

pelo CudaCanny são as mesmas identificadas pelo ItkCanny. Isso é especialmente relevante pois os critérios utilizados levam em consideração a localização exata dos pixels de borda, logo um deslocamento mínimo já não identificaria o pixel de borda como correto.

Outro dado que se pode reparar é que à medida em que são utilizadas imagens maiores nos testes, a taxa de acerto cresce enquanto a taxa de erro diminui. Isso se deve ao fato de a maioria dos erros de detecção do CudaCanny ocorrerem nos pixels localizados nas extremidades da imagem. Portanto, uma vez que imagens maiores possuem uma proporção maior de pixels internos do que de pixels localizados nas extremidades da imagem, a taxa de erro diminui nessa mesma proporção.

6.3 TESTES DE DESEMPENHO

Os resultados exibidos pelo gráfico da Figura 5.1 e pela Tabela 5.2 mostram que a execução do filtro Canny pelo CudaCanny possui tempo de execução muito inferior ao da implementação encontrada na biblioteca do ITK para todas as bases de imagens testadas. A diferença entre os tempos de execução em GPU e em CPU na maioria dos casos chega duas ordens de grandeza. Dessa forma foi necessário utilizar a escala logarítmica no eixo vertical para exibir os resultados de execução do CudaCanny e do ItkCanny. A média da soma dos tempos de execução do Canny do ITK em todas as imagens da base B4 totalizou mais de 30 minutos na CPU do *desktop* (C2D) e quase 14 minutos no CPU do servidor (Ci7). Enquanto que a média da soma dos tempos de execução do CudaCanny em todas as imagens da B4 totalizou pouco mais de 19 segundos na GPU G80, pouco mais de 8,5 segundos na GPU Tesla1 (GT200) e quase de 6,5 segundos na GPU Tesla2 (Fermi). O desvio padrão das execuções mostra que para cada execução as diferenças de *hardware* e de implementação representam ganhos reais de desempenho.

Os gráficos da Figura 5.2 mostram a comparação entre os tempos de execução do Canny para cada *hardware* em todas as bases. Esse gráfico foi dividido entre a comparação das execuções em placa gráfica e a comparação das execuções em placa gráfica com as CPUs, pois a diferença entre os resultados é muito grande. Enquanto que na comparação entre as placas gráficas o ganho de desempenho variou de 1,2 a 3,0 vezes, a comparação entre as placas gráficas e as CPUs variou de 25,5 a 278,9 vezes.

Na comparação de desempenho entre as GPUs, exibida pela Figura 5.2a, o melhor resultado ficou com a execução do CudaCanny pela Fermi chegando a ter desempenho 3,0 vezes melhor que a G80 para as duas maiores bases e de 2,3 e 2,7 vezes melhor, respectivamente,

para as bases B1 e B2. O melhor desempenho da Fermi pode ser explicado pela quantidade 4 vezes maior de processadores, pelo modelo de execução com 2 escalonadores, e pelas caches da memória global que reduzem o tempo de acesso ao esconder a latência de acesso.

Na comparação de desempenho com a GT200, a Fermi conseguiu desempenho de 1,27 a 1,33 vezes melhor. A diferença entre essas duas arquiteturas não foi muito expressiva uma vez que a GT200 já implementa diversas das flexibilizações de acesso a memória da Fermi além de contar com mais da metade dos núcleos de processamento da Fermi[2] (WHITEPAPER..., 2009; NVIDIA..., 2010b). A placa gráfica da arquitetura GT200, na comparação com a G80, conseguiu um desempenho variando de 1,8 a 2,9 vezes melhor devido a maior quantidade de núcleos e às flexibilizações de acesso à memória.

Os resultados da comparação de desempenho entre as placas gráficas e as CPUs, exibidos pela Figura 5.2b, mostraram que as comparações apresentaram uma curva ascendente a medida que são utilizadas bases maiores. Isso indica que o CudaCanny consegue processar conjuntos cada vez maiores de dados de forma mais eficiente que o ItkCanny em CPU.

A comparação com o melhor desempenho foi da placa gráfica com arquitetura Fermi sobre a CPU de dois núcleos (C2D) apresentando desempenho de 144,4 a 278,9 vezes melhor. Enquanto no desempenho por núcleos de processamento a Fermi leva vantagem pois apesar de possuir 224 vezes mais núcleos de processamento, o *clock* de seus núcleos possuem menos da metade da frequência dos núcleos da C2D. Na comparação com as 2 CPUs de 4 núcleos (Ci7), a Fermi atingiu desempenho de 60,8 a 129,8 vezes melhor. Comparando o desempenho com a proporção de processadores de cada arquitetura, mais uma vez a Fermi leva vantagem, pois possui 56 vezes mais núcleos cujo *clock* possui pouco mais de um terço da frequência.

A placa gráfica da arquitetura GT200 comparada à CPU C2D conseguiu desempenho de 112,8 a 210,5 vezes melhor. Mesmo possuindo 120 vezes mais núcleos, esse desempenho é vantajoso, pois para as execuções de maior volume de dados obtêm-se ganho de desempenho maior que a proporção de processadores. E mesmo para as demais execuções há de se considerar que os núcleos da GPU possuem menos da metade da frequência de *clock*. Comparada com a CPU do servidor (Ci7), o desempenho da GT200 foi de 47,5 vezes 97,6 vezes melhor. Na comparação com a proporção de processadores a GT200 também leva vantagem sobre a Ci7, pois ela possui 30 vezes mais processadores cujo *clock* corresponde a quase 40% da frequência dos núcleos da Ci7.

A comparação da placa gráfica (G80) com a CPU do computador *desktop*, a G80 mos-

[2] A placa gráfica da arquitetura GT200 testada possui 240 núcleos, enquanto que a placa gráfica da arquitetura Fermi possui 448.

trou desempenho de 60,7 a 91,8 vezes melhor. Comparando com a proporção de núcleos de processamento, a G80 possui 56 vezes mais núcleos com menor frequência de *clock*. E na comparação com a CPU Ci7, a G80 obteve resultados de 25,5 a 42,7 vezes melhor. Comparado com a proporção de núcleos de processamento, a G80 é vantajosa, pois possui 14 vezes mais processadores com quase a metade da frequência de *clock* e ainda assim atinge desempenho melhor.

Pode-se concluir que para todas as bases de dados e *hardware* testados as arquiteturas de placas gráficas obtiveram desempenho melhor, mesmo considerando a quantidade maior de núcleos de processamento. Esse ganho de desempenho deve-se ao modelo de paralelismo SIMD utilizado nas placas gráficas que permitiram aproveitar de forma mais eficiente o processamento da grande quantidade de núcleos encontrados nas placas gráficas.

O gráfico da Figura 5.3 mostra que as placas gráficas foram capazes de processar aproximadamente 100 vezes mais pixels por milissegundo que as CPUs nesse teste. Além disso, as curvas das placas gráficas entre as bases B1 e B3 e sua estabilização a partir da base B3 mostram que o desempenho do CudaCanny atinge um ponto de saturação próximo à quantidade de pixels da base B3. Dessa forma, a escalabilidade do modelo CUDA, ou seja, a capacidade de processar conjuntos maiores de dados em uma mesma quantidade de tempo, atinge seus limites físicos. A partir desse ponto de saturação é que obtêm-se os melhores desempenhos em GPU, como é possível observar também nos gráficos das Figuras 5.2a e 5.2b.

O gráfico da Figura 5.4a mostra que a maior parte do tempo de execução do CudaCanny é gasto na Histerese e ainda que a arquitetura Fermi consegue executar de forma mais otimizada a Histerese que as demais arquiteturas de placas gráficas.

As melhorias no acesso à memória das placas gráficas podem ser percebidas na diminuição da porcentagem de processamento gasto com a Histerese nas execuções do CudaCanny. As execuções do CudaCanny na arquitetura G80 utilizaram em média 47,7% do tempo na Histerese, enquanto as execuções da GT200 utilizaram 47% e da Fermi 42,4%. Os gráficos das Figuras 5.4a e 5.1, juntamente com os resultados dos Testes de Qualidade (ver Tabela 5.1), mostram que a abordagem utilizada para implementar a histerese em GPU, apesar de ser diferente da abordagem tradicional utilizada no ItkCanny, é capaz de retornar resultados muito próximos em termos de localização das bordas, com a vantagem de executar em muito menos tempo, graças ao paralelismo da GPU. Além disso, o gráfico mostra que há um ganho de desempenho considerável nas etapas da Gaussiana, das Derivadas e da NMS a medida que maiores volmes de dados são processados.

O gráfico da Figura 5.4b ainda mostra que a execução do ItkCanny consegue obter

algum ganho de desempenho ao aumentar a quantidade de núcleos de processamento, porém, esse ganho é restrito ao cálculo da Gaussiana, das Derivadas e da NMS. Enquanto que a Histerese não compartilha proporcionalmente desse ganho de desempenho aumentando a proporção de processamento gasto com a Histerese de 38,1% na C2D para 55,08% na Ci7. Dessa forma, a implementação da histerese no ItkCanny não consegue obter desempenho satisfatório ao aumentar a quantidade de núcleos, uma vez que com 4 vezes mais núcleos, as 2 CPUs de 4 núcleos consegue desempenho apenas 2,1 vezes melhor que a C2D.

6.4 TESTES DE ALGORITMO

Os resultados dos testes executados com os algoritmos descritos na seção 3.4 serão discutidos nas seções a seguir e mostram quais foram as estratégias mais eficientes.

6.4.1 Teste de Acessos à Memória

Para esse teste foi utilizado o algoritmo descrito na seção 3.4.1. A implementação mais simples desse algoritmo (1.A) apenas copia os elementos do vetor de entrada para o vetor de saída trocando a posição dos elementos ao escrever os elementos no vetor de saída. Como essas operações acessam diretamente a memória global, os acessos realizados para escrever os elementos no vetor de saída não obedecem ao princípio da coalescência descrito na seção 2.3.1.5. Consequentemente os acessos de cada *warp* à memória global são serializados por *thread* na arquitetura G80. Porém o desempenho nas arquiteturas GT200 e Fermi consegue ser melhor devido à flexibilização das regras de coalescência. Além disso a Fermi conta com uma cache que permite evitar a serialização de *warps* e otimizar acessos à memória.

Curiosamente, na arquitetura G80 os acessos não-coalecentes de leitura (1.B), ou desalinhados, foram mais eficientes que os acessos de escrita efetuados sob as mesmas circunstâncias (1.A). Isso fica claro nos tempos mostrados por esse teste na Tabela 5.3. Ao inverter as posições do vetor durante a leitura dos dados mostra que o acesso não-coalescente de escrita foi possível obter um desempenho 27% melhor. Porém nas arquiteturas mais recentes essa alteração não surtiu o mesmo efeito, obtendo tempos muito parecidos entre as implementações 1.A e 1.B.

Porém a estratégia que mostrou o melhor desempenho para o acesso desalinhado à memória é o uso da cache de textura (1.C). Nesse teste onde o acesso à memória influencia em quase todo o tempo de execução, um ganho de 88% de desempenho foi atingido na arquitetura G80 utilizando apenas a cache de textura para realizar a leitura desalinhada de vetor. Enquanto o ganho de desempenho na arquitetura GT200 foi de 48% e de 46% na Fermi. Lembrando

que a memória de textura é uma abstração da memória global que utiliza uma cache de textura. Portanto essa otimização não só visa reduzir o tempo de acesso através do *pre-fetch* da cache de textura, como também busca evitar os acessos não-coalescentes causados pela forma que a leitura e escrita na memória são realizadas, uma vez que a cache de textura não possui as restrições de acesso da memória global. Portanto além de fornecer um tempo menor de acesso à memória, a cache de textura possui a vantagem de acessar posições desalinhadas de memória de forma eficiente.

6.4.2 Teste de Serialização de Warps e Acessos à Memória

Este teste executou o algoritmo descrito na seção 3.4.2. Sua primeira implementação (2.A) utilizou instruções condicionais para decidir as posições que serão acessadas na memória global. Neste caso apenas em duas ocasiões ocorrem *warps* divergentes, que são os *warps* que acessam as primeiras e as últimas posições. Entretanto, os acessos realizados para preencher o valor das variáveis **val.x** e **val.y** são acessos desalinhados, uma vez que para *threads* com índice maior que zero **val.x** recebe o valor do índice anterior (**idx**-1) no vetor de entrada. Enquanto que para *threads* com índice menor que o tamanho dos vetores menos 1, **val.y** recebe o valor do índice anterior (**idx**+1) no vetor de entrada. Na excessão desses dois casos ambas variáveis recebem o valor do vetor de entrada de mesmo índice que sua *thread*. Portanto quase todos os acessos de leitura dessa implementação são desalinhados causando, nas placas G80, perda de desempenho. Na placa da arquitetura GT200, o desempenho é melhor devido à flexibilização das regras de coalescência. Enquanto que na Fermi o desempenho é beneficiado com o uso da cache.

A primeira otimização dessa implementação (2.B), utilizou a cache de textura para evitar os acessos não-coalescentes na leitura dos dados. Apenas essa mudança reduziu o tempo de execução do *kernel* em mais de 85% na placa de arquitetura G80. Na GT200 o ganho de desempenho dessa otimização ficou em aproximadamente 50%. Porém na arquitetura Fermi o uso da textura foi mais lento pois ao contrário das outras arquiteturas, a Fermi já contava com uma cache antes dessa otimização.

A segunda otimização eliminou as expressões condicionais da implementação e usou apenas expressões lógicas no cálculo das posições a serem acessadas pela cache de textura (2.C). Essa otimização mostrou-se com o melhor desempenho ao eliminar os *warps* divergentes que ocorriam nas extremidades dos vetores e conseguiu um ganho no desempenho de aproximadamente 23% na placa da arquitetura G80 e de 10% na GT200 e na Fermi. Uma vez que as arquiteturas GT200 e Fermi contam com acessos à memória global mais flexíveis e mais

multiprocessadores, o que beneficiou o desempenho da primeira implementação e reduziu os ganhos de desempenho das estratégias aplicadas. Nesse sentido, curiosamente, o uso da cache para a memória global na Fermi foi mais eficiente que o acesso através da cache de textura.

6.4.3 Teste de Serialização de Warp e Expressões Lógicas

O algoritmo da seção 3.4.3 foi utilizado na execução desse teste. Em sua implementação inicial (3.A), cada *warp* possui *threads* que divergem quanto à expressão condicional usada para definir os valores das variáveis **dir.x** e **dir.y**, portanto o *warp* executa todas as instruções do **if/else**. Enquanto as *threads* que avaliaram a condição como verdadeira continuam, as demais esperam. Isso se repete no **else** com as *threads* que avaliaram a condição como falsa executando enquanto as demais esperam. Evitando esse tipo de situação um *warp* inteiro pode executar apenas um lado da expressão condicional (o **if** ou o **else**) uma vez que todas as suas *threads* avaliem a condição da mesma forma.

A otimização realizada nesse algoritmo (3.B) eliminou o uso das expressões condicionais em detrimento de cálculos lógicos para calcular o valor das variáveis **dir.x** e **dir.y**. Essa estratégia permitiu reduzir o tempo de execução do algoritmo em 46% para a placa da arquitetura G80 testada, em 21% para a GT200 e em 41% para a Fermi. O desvio padrão para esse teste mostra que essa diferença é significativa, pois ele não é suficientemente alto para sustentar a afirmação de que o ganho de desempenho possa ter sido causado pela variação da medida dos tempos de execução.

Esse teste se diferencia do anterior por não focar nos tempos de acesso à memória, mas apenas no uso de expressões lógicas em detrimento das expressões condicionais. Enquanto no teste anterior a diferença entre as arquiteturas de GPU foi maior devido às melhorias de implementadas no acesso à memória global, esse teste mostra que as melhorias no modelo de execução, que incluem aumento na quantidade e capacidade dos multiprocessadores e a inclusão de um escalonador extra a partir da arquitetura Fermi, tiveram impacto menor, porém ainda significativo.

6.4.4 Teste de Configuração de Blocos e Grid

O algoritmo descrito na seção 3.4.4 não possui melhorias na implementação, mas apenas na configuração da execução de seu *kernel*. O *kernel* implementado executa apenas uma cópia de elementos entre dois vetores incrementando o valor dos elementos em 1. A ideia aqui é mostrar a diferença de desempenho causada por diferentes configurações de execução de um

kernel. A configuração de blocos e *grid* deve mostrar a preocupação com o paralelismo do *kernel*, pois esta influencia diretamente na ocupação dos multiprocessadores.

A execução do *kernel* deste teste com uma *grid* de 5 *threads* por bloco (4.A) prejudica a ocupação dos multiprocessadores, pois permite apenas 8 *warps* ativos, valor bem abaixo do máximo permitido em cada arquitetura[3]. Com 5 *threads* por bloco, cada bloco possui apenas um *warp*. Isso equivale a uma ocupação de *warps* por multiprocessadores de 33% na arquitetura G80, 25% na GT200 e 17% na Fermi, ou seja, apenas 40 *threads* ativas por multiprocessador de um máximo de 768 na arquitetura G80, 1024 na GT200 e 1536 na Fermi.

Além disso, essa configuração de *grid* utiliza blocos com menos *threads* que um *half-warp* (16 *threads*) causando serialização do acesso à memória, ou seja, acessos não-coalescentes. Nas arquiteturas GT200 e Fermi onde as regras de acesso à memória global são mais flexíveis, os acessos não são serializados, porém os acessos coalescentes são realizados por um *half-warp*, logo são realizados mais acessos que o necessário, pois, neste caso, os blocos são menores que um *half-warp*.

Uma boa dica é usar quantidades de *threads* múltiplas de 32, pois para todas as arquiteturas definem *warps* de 32 *threads*. Dessa forma é possível acessar a memória global para todas as *threads* de um *warp* em apenas uma única operação na arquitetura Fermi ou duas operações nas arquiteturas anteriores. Uma configuração adequada de bloco que é bastante utilizada contém 256 *threads* por bloco (4.B). Essa configuração permite maximizar a ocupação de *warps* por multiprocessador para todas as arquiteturas de GPU atuais. Todos os processadores nos multiprocessadores estarão ativos se cada *thread* utilizar no máximo 10, 16 ou 20 registradores nas arquiteturas G80, GT200 e Fermi respectivamente e se a quantidade de memória compartilhada por multiprocessador for distribuída corretamente pela quantidade de *warps* por multiprocessador. A correta configuração do *kernel* permitiu otimizar seu tempo de execução em 82% na arquitetura G80, em 84% na GT200 e em 89% na Fermi. Essa configuração também foi utilizada no CudaCanny e no CudaSobel em todos os *kernels*.

Neste teste foram utilizados blocos e *grid* unidimensionais. Porém o modelo CUDA de paralelismo permite que sejam criados *grids* de até 2 dimensões e blocos até 3 dimensões. Entretanto as regras de ocupação de multiprocessadores e de coalescência mantêm-se as mesmas.

[3]O limite físico é de 24 *warps* por multiprocessador na arquitetura G80. E de 32 *warps* nas placas das arquiteturas GT200 e Fermi utilizadas neste trabalho. Entretanto as placas mais novas da arquitetura Fermi possuem um limite de 48 *warps*

7 CONCLUSÃO

Neste trabalho foi apresentada a implementação do algoritmo de detecção de bordas Canny utilizando o modelo de programação em placas gráficas CUDA em uma classe para a biblioteca de processamento de imagens ITK. Essa implementação foi escolhida por ser um filtro de processamento de imagens composto por outros filtros. Para essa implementação foi utilizada a extensão CITK. Dessa forma foi possível entender os aspectos necessários à integração de um filtro que executa em placas gráficas em um fluxo de processamento de imagens do ITK e desenvolver uma classe de objeto ITK para armazenar as configurações dos *kernels* CUDA presentes no filtro.

O uso de placas gráficas no processamento de imagens é importante devido ao interesse em aproveitar o desempenho das arquiteturas paralelas para reduzir o tempo de execução dos filtros de processamento de imagens do ITK. Através deste trabalho foi possível identificar as necessidades que permitem o desenvolvimento de ferramentas dentro do ambiente do ITK para facilitar o uso dos modelos de paralelismo atuais.

Outro ponto importante é a confirmação da ganho de desempenho alcançado pela implementação de um filtro de processamento de imagens através de um modelo de processamento paralelo em placas gráficas mantendo a mesma qualidade de detecção de bordas. Em todos os testes realizados, a implementação do filtro de detecção de bordas Canny em CUDA foi mais rápida que a implementação do Canny encontrada no ITK.

Uma das desvantagens da utilização do modelo de programação em placas gráficas é a dificuldade em implementar operações que dependam de regiões não contíguas de uma imagem (como foi o caso da histerese), pois neste caso, o acesso é serializado. Entretanto, o CudaCanny mostrou que é possível obter melhoras bastante consideráveis de desempenho utilizando abordagens alternativas. Além disso, a arquitetura Fermi reduziu este problema com a implementação de uma cache para a memória global. Mas a necessidade de realizar cópias entre as memórias ainda pode limitar o desempenho de algumas aplicações.

Devido à importância de permitir ao programador de filtros utilizar o processamento

em placas gráficas e o ganho de desempenho que se pode atingir para diferentes aplicações demonstrados nesse trabalho, os desenvolvedores do ITK estão empenhados em desenvolver uma solução que permita esse tipo de processamento ser realizado no ambiente ITK da maneira mais eficiente.

7.1 TRABALHOS FUTUROS

Entre as possibilidades abertas com este trabalho, pode-se destacar a reimplementação do CudaCanny utilizando a classe itkTileImageFilter contida na extensão CITK para permitir que imagens maiores que 4096×4096 possam ser processadas. Outra melhoria ao CudaCanny poderia torná-lo mais genérico, uma vez que a atual implementação permite apenas o processamento com pixels em **float**.

Melhorias no filtro itkCudaInterface podem implementar novas funcionalidades como o gerenciamento de multi-GPUs, referências de textura, configurações de memória compartilhada e cache, *streams*. Dessa forma será possível unificar o gerenciamento de configurações de *kernel* em uma estrutura própria do ITK.

A possibilidade de utilizar outras tecnologias como OpenMP, SSE, APU, ou ainda OpenCL podem ampliar as possibilidades de paralelismo em filtros do ITK e consequentemente melhorar o desempenho de muitas aplicações.

E, por fim, o estudo da comparação de modelos de programação paralela em placas gráficas e CPUs multi-core com o modelo de programação sequencial se faz cada vez mais necessário em um contexto onde a programação paralela é cada vez mais presente.

REFERÊNCIAS BIBLIOGRÁFICAS

ARBELAEZ, P.; FOWLKES, C.; MARTIN, D. *The Berkeley Segmentation Dataset and Benchmark*. 2007. Acesso em: 29/11/2010. Disponível em: <http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/>.

BANKMAN, I. H. *Handbook of Medical Image Processing and Analysis*. 2. ed. [S.l.]: Elsevier, 2009.

BEARE, R. et al. *CITK - an architecture and examples of CUDA enabled ITK filters*. May 2011. Acesso em: 25/05/2011. Disponível em: <http://hdl.handle.net/10380/3269>.

BOAVENTURA, I. A. G.; GONZAGA, A. Método de avaliação de detector de bordas em imagens digitais. *Anais do V Worskhop de Visão Computacional*, 2009.

BORKAR, S. Design challenges of technology scaling. *IEEE Micro*, IEEE Computer Society, Los Alamitos, CA, USA, v. 19, p. 23–29, 1999. ISSN 0272-1732.

BRÄUNL, T. et al. *Parallel Image Processing*. Berlin; New York, NY: Springer, 2001.

CANNY, J. *Finding Edges and Lines in Images*. [S.l.], 1983. Department of Eletrical Engeneering and Computer Science, Massachusetts Institute of Tecnology, Cambridge. Relatório Técnico.

CANNY, J. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 8, n. 6, p. 679–698, 1986. ISSN 0162-8828.

CROSS Plataform Make. 2000. Acesso em: 11/02/2010. Disponível em: <http://www.cmake.org>.

CUDA Insight Toolkit. 2010. Acesso em: 10/08/2010. Disponível em: <http://code.google.com/p/cuda-insight-toolkit/>.

CUDA Zone. 2007. Acesso em: 10/03/2009. Disponível em: <http://www.nvidia.com/object/cuda_home.html>.

FARBER, R. *CUDA, Supercomuting for the Masses*. 2008–2010. Acesso em: 22/12/2009. Disponível em: <http://www.ddj.com/hpc-high-performance-computing/207200659>.

FUNG, J.; MANN, S. Openvidia: parallel gpu computer vision. In: *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*. New York, NY, USA: ACM, 2005. p. 849–852. ISBN 1-59593-044-2.

GENERAL-PURPOSE Computation on Graphics Hardware. 2002. Acesso em: 05/09/2009. Disponível em: <http://www.gpgpu.org>.

GHULOUM, A. Viewpoint face the inevitable, embrace parallelism. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 9, p. 36–38, 2009. ISSN 0001-0782.

GÓMEZ-LUNA, J. et al. Parallelization of a video segmentation algorithm on cuda—enabled graphics processing units. In: *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009. p. 924–935. ISBN 978-3-642-03868-6.

GURCAN, M. N. et al. Computerized pathological image analysis for neuroblastoma prognosis. *Proceedings of the 2007 American Medical Informatics Association Annual Symposium (2007)*, p. V –525–V –528, 2007. ISSN 1522-4880.

HARTLEY, T. D. et al. Biomedical image analysis on a cooperative cluster of gpus and multicores. In: *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008. p. 15–25. ISBN 978-1-60558-158-3.

IBÁÑEZ, L. et al. *The ITK Software Guide*. [S.l.]: Kitware, Inc., Jan 2003. E-book.

IBÁÑEZ, L. et al. *The ITK Software Guide Second Edition Updated for ITK version 2.4*. November 21 2005. E-book.

INSIGHT Segmentation and Registration ToolKit. 1999. Acesso em: 02/08/2009. Disponível em: <http://www.itk.org>.

ITK Image Registration with CUDA. 2008. Acesso em: 17/06/2009. Disponível em: <http://wiki.na-mic.org/Wiki/index.php/ITK_Image_Registration_with_CUDA>.

ITK Release 4/GPU Acceleration. 2010–2011. Acesso em: 04/02/2011. Disponível em: <http://www.itk.org/Wiki/ITK_Release_4/GPU_Acceleration>.

JEONG, W.-K. *CUDAITK*. 2007. Acesso em: 09/10/2009. Disponível em: <http://www.cs.utah.edu/ wkjeong/>.

KIDWAI, H. K.; SIBAI, F. N.; RABIE, T. F. Parallelization and performance evaluation of an edge detection algorithm on a streaming multi-core engine. *JITR*, v. 2, n. 4, p. 81–91, 2009.

LINDEBERG, T. Edge detection and ridge detection with automatic scale selection. *Int. J. of Computer Vision*, v. 30, 1998.

LUO, Y.; DURAISWAMI, R. Canny edge detection on nvidia cuda. *Computer Vision and Pattern Recognition Workshop*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1–8, 2008.

MARTIN, D. et al. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: *Proc. 8th Int'l Conf. Computer Vision*. [S.l.: s.n.], 2001. v. 2, p. 416–423.

NICKOLLS, J. et al. Scalable parallel programming with cuda. *Queue*, ACM, New York, NY, USA, v. 6, n. 2, p. 40–53, 2008. ISSN 1542-7730.

NVIDIA CUDA C Getting Started for Linux. August 19 2010. E-book.

NVIDIA CUDA C Programming Guide. Version 3.2. September 11 2010. E-book.

OPENCL. 2008. Acesso em: 17/03/2010. Disponível em: <http://www.khronos.org/opencl>.

PIEPER, S. et al. The na-mic kit: Itk, vtk, pipelines,grids and 3d slicer as an open platform for the medical image computing community. In: *In Proceedings of the Third IEEE International Symposium on Biomedical Imaging (ISBI '06)*. [S.l.: s.n.], 2006.

PODLOZHNYUK, V. *Image Convolution with CUDA*. June 2007. Acesso em: 12/03/2010. Disponível em: <http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf>.

PROPOSALS:GPU_INTEGRATION. 2007–2009. Acesso em: 12/03/2010. Disponível em: <http://www.itk.org/Wiki/Proposals:GPU_Integration>.

STILL, M. *The Definitive Guide to ImageMagick (Definitive Guide)*. Berkely, CA, USA: Apress, 2005. ISBN 1590595904.

STONE, S. S. et al. How gpus can improve the quality of magnetic resonance imaging. In: *In The First Workshop on General Purpose Processing on Graphics Processing Units*. [S.l.: s.n.], 2007.

WHITEPAPER NVIDIA's Next Generation CUDA Compute Architecture Fermi. 2009. E-book.